Разработка геоприложений на языке Python

Эрик Вестра

[PACKT] PUBLISHING

ДМК издательство

Эрик Вестра

Разработка геоприложений на языке Python

Erik Westra

Python Geospatial Development

Develop sophisticated mapping applications from scratch using Python 3 tools for geospatial development

Third Edition

[PACKT] PUBLISHING
BIRMINGHAM – MUMBAI

open source
community experience distilled

Эрик Вестра

Разработка геоприложений на языке Python

Разработка сложных картографических приложений с нуля с использованием геоинструментальных средств Python 3

Третье издание

Москва, 2017

УДК 528.92:004.9Python
ББК 26.17с
В38

Вестра Э.
В38 Разработка геоприложений на языке Python / пер. с англ. А. В. Логунова. – М.: ДМК Пресс, 2017. – 446 с.: ил.

ISBN 978-5-97060-437-3

Написание геопространственных программ предполагает решение таких задач, как группирование данных по географическому положению, хранение и анализ больших массивов информации, выполнение сложных расчетов и построение красочных интерактивных карт.

Книга предоставляет обзор главных геопространственных понятий, источников геоданных и наборов инструментов для геообработки. Рассмотрены приемы хранения и доступа к пространственным данным. Показано создание собственного интерфейса со скользящей картой в рамках веб-приложения. Подробно описано создание редактора геоданных на основе географического модуля GeoDjango для веб-платформы Django.

Издание адресовано опытным разработчикам на языке Python, которые хотели бы освоить концепции геопрограммирования, методы получения и работы с геоданными, решать пространственные задачи и конструировать сложные картографические приложения.

УДК 528.92:004.9Python
ББК 26.17с

ISBN 978-1-78528-893-7 (анг.)
ISBN 978-5-97060-437-3 (рус.)

Содержание

Об авторе

Эрик Вестра уже больше 25 лет является профессиональным разработчиком программного обеспечения, который в течение последнего десятилетия работает почти исключительно с Python. Первоначальный интерес Эрика к проектированию графического интерфейса пользователя привел к разработке одной из самых продвинутых систем срочной курьерской доставки, используемых службами и компаниями курьерской доставки по всему миру. В последние годы Эрик участвует в разработке и реализации систем по подбору поставщиков для потребителей товаров и услуг по целому ряду регионов с широкой географией и с использованием разных систем обмена мгновенными сообщениями и платежных систем. Эта работа включает в себя создание геокодеров в реальном масштабе времени и режимов просмотра постоянно меняющихся данных географических карт. Эрик проживает в Новой Зеландии и работает на компании, расположенные по всему миру.

Кроме того, он является автором выпущенных в издательстве Packt книг «*Геоанализ на Python*» и «*Создание картографических приложений*» на основе QGIS, а также предстоящей публикации «*Модульное программирование на языке Python*».

Я хотел бы поблагодарить Рут за ее великолепие, а моих дочерей – за их терпение. Без вас ничего из этого не было бы возможным.

О рецензенте

Лу Може получил образование в области информатики очень давно в Университете штата Мичиган, где учился использовать программное обеспечение для разработки циклотрона. Затем он в течение 34 лет проработал в компании IBM и потом продолжил карьеру, работая на несколько консалтинговых фирм, включая долгосрочное сотрудничество с железнодорожной отраслью индустрии. В настоящее время он консультирует компанию Keyhole Software, расположенную в Ливуд, шт. Канзас. Прошлой весной для этой компании он создал программный инструмент MockOlap для каркасного прототипирования на основе операций перетаскивания. Лу программирует на C++, Java и более новых языках и в настоящее время интересуется микросервисами, Docker, Node.js, NoSQL, геопространственными системами, функциональным программированием, мобильными и одностраничными веб-приложениями – любым новым языком или платформой. Лу иногда ведет блог, посвященный программным технологиям. Он является соавтором трех книг по информатике, написал два учебных руководства по XML для IBM DeveloperWorks и учебное руководство по LDAP для журнала WebSphere. Кроме того, Лу является соавтором нескольких сертификационных тестов по J2EE для IBM, а также рецензентом ряда других издателей.

Предисловие

С ростом популярности картографических веб-сайтов и пространственно ориентированных устройств и приложений методы разработки геопространственного программного обеспечения образовали быстро развивающуюся область информатики – геоинформатику. Будучи разработчиком на Python, вы не можете позволить себе отстать от новейших тенденций. В сегодняшнем информированном о местоположении мире каждый разработчик на Python может извлечь выгоду из понимания концепций и методики разработки геопространственных приложений.

Работа с геопространственными данными бывает непростой, потому что она связана с математическими моделями земной поверхности. Но поскольку Python – это мощный язык программирования с большим количеством высокоуровневых программных инструментов, он идеально подходит для разработки геопространственных приложений. Эта книга познакомит вас с инструментами языка Python, которые требуются для разработки геопространственных приложений. Она проведет вас по ключевым геопространственным понятиям, таким как географическое положение, расстояние, единицы измерения, картографические проекции, геодезические датумы и форматы геопространственных данных. Затем мы займемся изучением ряда программных библиотек Python и воспользуемся ими и общедоступными геопространственными данными для решения самых разнообразных задач. Книга предоставляет углубленный анализ методов хранения пространственных данных в базе данных и приемов использования пространственных баз данных в качестве инструментов для решения широкого диапазона геопространственных задач.

В ней подробно рассматриваются методы генерирования карт при помощи инструмента визуализации цифровых карт – программной библиотеки Mapnik. Кроме того, книга поможет вам создать высокотехнологичное геопространственное веб-приложение с функционалом редактирования карты на основе географического модуля GeoDjango для веб-платформы Django, программной библиотеки Mapnik и геопространственного расширения PostGIS для СУБД PostgreSQL. К концу книги вы научитесь интегрировать пространственный функционал в свои собственные приложения и создавать полнофункциональные картографические приложения с нуля.

Эта книга представляет собой практическое руководство, которое научит вас приемам получения доступа к геоданным, управления ими и их визуализации, используя широкий диапазон инструментов Python для разработки геоинформационных систем (ГИС).

О чем эта книга рассказывает

Глава 1 «Разработка геопрограмм на Python» предлагает обзор языка программирования Python и концепций, лежащих в основе процесса разработки геопро-

странственного приложения. Кроме того, будут затронуты основные прецеденты использования разработок геопространственных приложений, последние достижения и тенденции ближайшего будущего в данной области.

Глава 2 «*Геоинформационные системы*» посвящена ознакомлению с базовыми понятиями, включая географическое положение, расстояние, единицы измерения, картографические проекции, географические фигуры, геодезические датумы и форматы геоданных, и затем обсуждению процесса работы с геоданными в ручном режиме.

Глава 3 «*Библиотеки Python для геопрограммирования*» разбирает основные библиотеки Python, предназначенные для разработки геоприложений, в том числе функционал библиотек, способы их установки, важные понятия, необходимые для понимания работы библиотек, и способы их применения.

Глава 4 «*Источники геоданных*» посвящена исследованию главных общедоступных источников геоданных, характеристике имеющейся в распоряжении информации, используемым форматам данных и приемам импортирования данных после их скачивания.

Глава 5 «*Решение задач с геоданными на Python*» посвящена применению ранее представленных библиотек для выполнения различных задач с использованием геоданных, включая смену картографических проекций, импорт и экспорт данных, конвертирование и стандартизацию единиц геометрий и расстояний, выполнение геопространственных расчетов.

Глава 6 «*Пространственные базы данных*» вводит понятия, лежащие в основе пространственных баз данных, и затем подробно рассматривает расширение PostGIS для СУБД PostgreSQL с поддержкой пространственных данных, способы его установки и использования из программы на Python.

Глава 7 «*Генерирование карт при помощи Python и библиотеки Mapnik*» предлагает подробный анализ инструмента картографирования Mapnik и приемов его применения для генерирования разнообразных карт.

Глава 8 «*Работа с пространственными данными*» познакомит с процессом проектирования и реализации полнофункционального геоприложения под названием DISTAL, используя общедоступные геоданные, хранящиеся в пространственной базе данных.

Глава 9 «*Совершенствование приложения DISTAL*» посвящена развитию функционала приложения из предыдущей главы для решения ряда задач, связанных с удобством использования и производительностью.

Глава 10 «*Инструменты для разработки геопространственных веб-приложений*» исследует понятия платформ веб-приложений, веб-служб, библиотек пользовательского интерфейса на JavaScript и скользящих карт. Данная глава познакомит с рядом стандартных веб-протоколов, используемых в геопространственных приложениях, и закончится обзором инструментария, при помощи которого в трех заключительных главах книги будет создано полнофункциональное картографическое приложение.

Глава 11 «*Собираем все вместе – полнофункциональная картографическая система*» знакомит с полноценным и высокотехнологичным веб-приложением

ShapeEditor, созданным с использованием геопространственного расширения PostGIS, библиотеки Mapnik и географического модуля GeoDjango. Работа начинается с проектирования законченного приложения и затем продолжается созданием моделей базы данных приложения ShapeEditor.

Глава 12 «ShapeEditor – импорт и экспорт файлов фигур» продолжает реализацию системы ShapeEditor, концентрируясь на отображении списка импортированных файлов фигур и подпрограммах импорта и экспорта файлов фигур через веб-браузер.

Глава 13 «ShapeEditor – выбор и правка геообъектов» завершает реализацию системы ShapeEditor добавлением подпрограмм, которые предлагают пользователю функционал для выбора и редактирования геообъектов внутри импортированного файла фигур. Он включает в себя создание специального сервера сборных цифровых карт и использование картографической библиотеки OpenLayers на JavaScript для визуализации геоданных на экране компьютера и взаимодействия с ними.

Что требуется для этой книги

Третье издание данной книги было расширено с целью поддержки Python 3, хотя, если хотите, вы можете продолжить пользоваться Python 2. Вам также понадобятся следующие инструменты и библиотеки, которые следует скачать и установить; подробные инструкции даны в соответствующих разделах книги:

- программный пакет **GDAL/OGR**;
- динамическая библиотека **GEOS**;
- библиотека Python **Shapely**;
- динамическая библиотека **Proj**;
- библиотека Python **pyproj**;
- СУБД **PostgreSQL** + программа администрирования **pgAdmin III**;
- геопространственное расширение **PostGIS** для СУБД PostgreSQL;
- адаптер СУБД PostgreSQL **psycopg2** для Python;
- динамическая библиотека **Mapnik**;
- веб-платформа **Django**;
- географический модуль **GeoDjango** для веб-платформы Django;
- картографическая библиотека **OpenLayers** на JavaScript.

Для кого эта книга

Эта книга предназначена для опытных разработчиков на языке Python, которые хотели бы освоить свободно распространяемый инструментарий и методику разработки геопространственных приложений с целью создания своих собственных геоприложений либо интеграции геопространственной технологии в свои существующие программы на Python.

Условные обозначения

В этой книге вы найдете ряд текстовых стилей, которые выделяют различные виды информации. Вот некоторые примеры этих стилей и объяснение их значения.

Фрагменты программного кода в тексте показаны следующим образом: «Набор данных `gdal.Dataset` представляет файл, который содержит данные в растровом формате».

Блок кода выглядит следующим образом:

```
import pyproj

lat1,long1 = (37.8101274,-122.4104622)
lat2,long2 = (37.80237485,-122.405832766082)

geod = pyproj.Geod(ellps="WGS84")
angle1,angle2,distance = geod.inv(long1, lat1, long2, lat2)

print("Расстояние равно {:0.2f} метров".format(distance))
```

Когда требуется привлечь ваше внимание к конкретной части блока кода, соответствующие строки или элементы выделяются полужирным шрифтом:

```
for value in values:
    if value != band.GetNoDataValue():
        try:
            histogram[value] += 1
        except KeyError:
            histogram[value] = 1
```

Любой ввод команды из командной строки или вывод результатов их вычисления оформляется следующим образом:

```
% python3 calcBoundingBoxes.py
Afghanistan (AFG) lat=29.4061..38.4721, long=60.5042..74.9157
Albania (ALB) lat=39.6447..42.6619, long=19.2825..21.0542
Algeria (DZA) lat=18.9764..37.0914, long=-8.6672..11.9865
```

Новые термины и **важные слова** показаны полужирным шрифтом. Слова, которые вы видите на экране, например в меню или диалоговых окнах, выглядят в тексте следующим образом: «Нажмите на гиперссылке **Download Domestic Names**, чтобы скачать национальные географические названия».

Предупреждения или важные примечания появляются в этом поле.

Подсказки и приемы появляются тут.

Дополнения к тексту оригинала книги.

Отзывы читателей

Отзывы наших читателей всегда приветствуются. Сообщите нам, что вы думаете об этой книге – что вам понравилось, а что нет. Обратная связь с читателями для нас очень важна, поскольку она помогает нам формировать названия книг, из которых вы действительно получите максимум полезного.

Отзыв по общим вопросам можно отправить по адресу feedback@packtpub.com, упомянув заголовок книги в теме вашего электронного сообщения.

Если речь о теме, в которой у вас есть экспертные знания, и вы интересуетесь написанием либо содействием в написании книги, то обратитесь к нашему перечню авторов по адресу www.packtpub.com/authors.

Служба поддержки

Теперь, когда вы являетесь довольным владельцем книги издательства Packt, мы предложим вам ряд возможностей с целью помочь вам получать максимум от своей покупки.

Скачивание исходного кода примеров

Вы можете скачать файлы с кодом примеров из вашего аккаунта по адресу http://www.packtpub.com/ для всех книг издательства Packt Publishing, которые вы приобрели. Если вы купили эту книгу в другом месте, то можно посетить http://www.packtpub.com/support и зарегистрироваться там, чтобы получить файлы прямо по электронной почте.

Загрузить файлы с примерами программного кода можно, выполнив следующие шаги:

1. Войдите на наш веб-сайт или зарегистрируйтесь там, используя ваш адрес электронной почты и пароль.
2. Наведите указатель мыши на вкладку **SUPPORT** вверху страницы.
3. Щелкните по разделу **Code Downloads & Errata**, посвященному примерам программного кода и опечаткам.
4. Введите название книги в **Поле поиска**.
5. Выберите книгу, для которой вы хотели бы скачать файлы с примерами.
6. Из раскрывающего меню выберите место, где вы купили эту книгу.
7. Щелкните по **Code Download**, чтобы скачать примеры.

Кроме того, файлы примеров можно скачать, нажав на кнопку **Code Files** на странице книги на веб-сайте издательства Packt Publishing. Доступ к этой странице можно получить, введя наименование книги в **Поле поиска**. Обращаем внимание, что для этого вы должны войти в ваш аккаунт в издательстве Packt.

Скачав файл, пожалуйста, убедитесь, что вы разархивировали или извлекли папку, воспользовавшись последней версией указанных ниже архиваторов:

- WinRAR / 7-Zip для Windows;
- Zipeg / iZip / UnRarX для Mac;
- 7-Zip / PeaZip для Linux.

Помимо этого, пакет примеров программного кода, прилагаемый к данной книге, размещен на GitHub по адресу https://github.com/PacktPublishing/Python-Geospatial-Development-Third-Edition. Мы также располагаем другими пакетами примеров программного кода, которые можно выбрать из нашего богатого каталога книг и видео, предлагаемого на странице https://github.com/PacktPublishing/. Можете убедиться сами!

Опечатки

Несмотря на то что мы приняли все меры, чтобы гарантировать корректность нашего информационного материала, ошибки действительно случаются. Если вы найдете ошибку в одной из наших книг, возможно, в тексте или коде, то мы будем благодарны, если вы сообщите об этом нам. Поступая так, вы можете уберечь других читателей от разочарования и помочь нам улучшать последующие версии этой книги. Если вы найдете какие-либо опечатки, пожалуйста, сообщите о них, посетив страницу http://www.packtpub.com/submit-errata, выбрав вашу книгу, нажав на ссылку формы **errata submission form** для предоставления информации об опечатке и введя ваши данные об опечатках. Как только ваши данные будут проверены, сообщение будет принято к рассмотрению, и данные об опечатке будут загружены на наш веб-сайт или добавлены к любому списку из существующих списков опечаток в разделе Errata под соответствующим заголовком.

Чтобы просмотреть ранее предоставленные опечатки, перейдите на страницу https://www.packtpub.com/books/content/support и в поле поиска введите название книги. Запрошенная информация появится под разделом Errata.

Нарушение авторских прав

Пиратство защищенного авторским правом материала в Интернете является хронической проблемой во всех средствах массовой информации. В издательстве Packt мы очень серьезно относимся к защите нашего авторского права и лицензий. Если вы сталкиваетесь с какими-либо недопустимыми копиями наших работ в какой-либо форме в Интернете, пожалуйста, просим вас незамедлительно предоставить нам адрес размещения или название веб-сайта, чтобы мы могли добиваться правовой защиты.

Пожалуйста, свяжитесь с нами по электронному адресу copyright@packtpub.com со ссылкой на предполагаемый пиратский материал.

Мы ценим вашу помощь в защите наших авторов и в наших усилиях предоставлять вам ценный информационный материал.

Вопросы

Если у вас есть вопрос по каким-либо аспектам этой книги, то вы можете связаться с нами по электронному адресу questions@packtpub.com, и мы приложим все усилия, чтобы решить ваш вопрос.

Комментарий переводчика

Весь материал книги приведен в соответствие с последними действующими версиями библиотек (время перевода книги – август-сентябрь 2016 г.), дополнен свежей информацией и протестирован в среде Windows 10 и Fedora 24. При тестировании программного кода за основу взят Python версии 3.5.2.

Книга содержит много аббревиатур и технических терминов из разных областей науки. Для удобства большинство аббревиатур кратко определено в сносках, а для некоторых терминов в силу отсутствия единой терминологии приведены соответствующие варианты наименований или пояснения. В конце книги основные термины и аббревиатуры собраны в *глоссарии*.

Поскольку в оригинале книги и, главное, во всех инструментальных средах программирования для отделения целой части числа от дробной используется не запятая, а точка, и чтобы не вносить путаницу при работе с книгой, в переводе оставлена форма записи чисел как есть, без изменений.

Книга может быть интересной широкому кругу специалистов, в том числе начинающим аналитикам данных, преподавателям, студентам, а также всем, кто интересуется геопрограммированием.

Далее приведены особенности установки и работы некоторого используемого программного обеспечения.

Установка и настройка дистрибутива Anaconda

Anaconda — это полностью свободный дистрибутив Python (предназначенный в том числе для коммерческого использования и повторного распространения). Он содержит более 400 самых популярных библиотек Python для вычислений в области естественных наук, математики, инженерии и анализа данных.

Установка дистрибутива в Windows выполняется стандартным образом после загрузки с сайта бинарного файла дистрибутива. В Linux сначала необходимо скачать установщик — скриптовый файл с расширением .sh для оболочки Bash (https://www.continuum.io/downloads#_unix). На примере Anaconda 4.2.0 (с Python 3.5.2) для 64-разрядных операционных систем команда установки выглядит так: `bash Anaconda3-4.2.0-Linux-x86_64.sh`.

Отметим, что инсталляция вступит в силу после того, как вы закроете и заново откроете окно терминала.

Для обновления дистрибутива Anaconda следует набрать в терминале следующую команду:

```
conda update conda
```

Удаление дистрибутива Anaconda:

```
rm -rf ~/anaconda
```

Более подробная информация по деинсталляции Anaconda содержится по указанной выше ссылке.

Чтобы удостовериться, что дистрибутив Anaconda установлен успешно, воспользуйтесь следующей командой:

```
conda --version
```

В результате будет выведен номер установленной версии.

В случае если вы в основном работаете в среде Python 3, но иногда возникает необходимость переключиться в среду Python 2 и использовать ее для работы с библиотеками, которые предназначены только для Python 2, то Anaconda предлагает функционал создания и активации новой среды, куда можно установить нужную версию языка. В нижеследующей таблице приведено несколько команд, которые можно выполнить прямо в Spyder во встроенном окне командной строки (**Инструменты ⇨ Открыть командную строку**):

`conda create -n py27 python=2.7.12 anaconda`	Установить другую версию Python (2.7.12) в новую среду с именем py27 (имя может быть любым)
`conda info --envs`	Проверить, что среда с именем py27 установлена (команда выводит список всех сред, при этом активная среда выделяется знаком *)
`source activate py27` (Linux, OS X), `activate py27` (Windows)	Переключиться в среду py27 с другой версией Python (команда activate добавляет в начало строки путь к среде py27)
`deactivate`	Деактивировать текущую среду и вернуться к среде по умолчанию
`python --version`	Проверить, что среда использует нужную версию Python
`conda remove -n py27 --all`	Удалить среду py27 (после выполнения команды выполнить conda clean --lock)
`conda install --name py27 scypy`	Установить библиотеку scypy в среду py27
`conda remove --name py27 scypy`	Удалить библиотеку scypy в среде py27
`conda clean --lock`	Очистить блокировку, если произошел сбой при установке среды (иногда сперва требуется удалить conda в Диспетчере задач)

Установка и настройка инструментальной среды Spyder

Spyder (в октябре 2016 г. появилась версия 3.0.0 с локализованным на русский язык графическим интерфейсом) — это инструментальная среда для научных вычислений для языка Python (Scientific PYthon Development EnviRonment) для Windows, Mac OS X и Linux. Это простая, легковесная и бесплатная интерактив-

ная среда разработки на Python, которая предлагает функционал, аналогичный среде разработки на MATLAB, включая готовые к использованию виджеты PyQt5 и PySide: редактор исходного кода, редактор массивов данных NumPy, редактор словарей, консоли Python и IPython и многое другое.

Для пользователей Windows хорошая новость заключается в том, что Spyder уже включен в состав дистрибутива Anaconda Python, и исполнимый файл находится в папке C:\Users\[ИМЯ_ПОЛЬЗОВАТЕЛЯ\Anaconda3\Scripts\spyder.exe. Чтобы установить среду Spyder без установки Anaconda, необходимо скачать соответствующий файл whl (https://pypi.python.org/pypi/spyder) и установить, как объяснено в разделе *Установка библиотек Python из whl-файла* ниже. Чтобы установить среду Spyder в Ubuntu Linux, используя официальный менеджер пакетов, нужна всего одна команда:

```
sudo apt-get install spyder3
```

Чтобы установить с использованием менеджера пакетов pip:

```
sudo apt-get install python-qt5 python-sphinx
sudo pip3 install spyder
```

И чтобы обновить:

```
sudo pip3 install -U spyder
```

Установка среды Spyder в Fedora 24 приведена ниже в разделе *«Подготовка среды Python 3 в ОС Fedora 24»*. Во всех вышеперечисленных случаях речь идет о версии Spyder для Python 3 (на момент инсталляции это был Python 3.5.2). Чтобы установить версию Spyder для Python 2, нужно просто поменять spyder3 на spyder.

Настройка среды Spyder с Python 3 для работы с Python 2

При инсталляции дистрибутива Anaconda3 по умолчанию базовым является интерпретатор Python 3 (в данном случае версии 3.5.2). В случае если нужно на время переключиться в режим работы в интерпретаторе Python версии 2 (в данном случае версии 2.7.12), существуют два варианта: дополнительно инсталлировать дистрибутив Anaconda для Python 2 либо выполнить небольшую настройку инструментальной среды Spyder, по умолчанию работающей на основе Python 3.

Для такой настройки нужно в основном меню выбрать **Инструменты** и затем **Параметры**. В открывшемся окне **Параметры** выбрать **Интерпретатор Python**. В разделе **Интерпретатор Python** с двумя переключателями установить переключатель **Использовать следующий интерпретатор Python** и затем нажать кнопку напротив текстового поля, чтобы выбрать путь к интерпретатору Python 2.7.12, который находится в папке C:\Users*[имя_пользователя]*\Anaconda3\envs\py27, где py27 — это имя среды, созданной вами для Python 2.7.12 (см. раздел *«Настройка дистрибутива Anaconda»* ранее). Если открыть новую консоль (**Консоли ⇨ Открыть консоль Python**), то в строке приветствия будет указана нужная версия Python.

Теперь в этой консоли можно набирать команды и вставлять целые программы, однако запускать программы из рабочего окна редактора Spyder не получится. Для

сценария, работающего с Python 2, требуется еще одна, и последняя, настройка. В основном меню нужно выбрать **Запуск**, затем **Настроить** и в разделе **Консоль** выбрать второй переключатель **Выполнить во внешнем системном терминале** (то есть в новой специально выделенной консоли). Теперь этот конкретный спенарий будет выполняться в консоли с Python 2.

Закончив работу с интерпретатором Python 2.7.12, следует не забыть вернуться к исходному интерпретатору Python 3.5.2. Для этого нужно в окне **Интерпретатор Python** просто установить переключатель **По умолчанию** (тот же, что и для Spyder).

Напомним, что месторасположение базового интерпретатора следующее: `C:\Users\[ИМЯ_ПОЛЬЗОВАТЕЛЯ]\Anaconda3\python.exe`. Разумеется, такой режим работы вносит некоторые неудобства, но, по крайней мере, он не такой громоздкий, как установка еще одной версии Anaconda 4.2.0, но уже с Python 2.7.12 в качестве базового интерпретатора.

Установка библиотек Python из whl-файла

Библиотеки для Python можно разрабатывать не только на чистом Python. Довольно часто библиотеки пишутся на C (динамические библиотеки), и для них пишется обертка Python, или же библиотека пишется на Python, но для оптимизации узких мест часть кода пишется на C. Такие библиотеки получаются очень быстрыми, однако библиотеки с вкраплениями кода на C программисту на Python тяжелее установить ввиду банального отсутствия соответствующих знаний либо необходимых компонентов и настроек в рабочей среде (в особенности в Windows). Для решения описанных проблем разработан специальный формат (файлы с расширением .whl) для распространения библиотек, который содержит заранее скомпилированную версию библиотеки со всеми ее зависимостями. Формат whl поддерживается всеми основными платформами (Mac OS X, Linux, Windows).

Установка производится с помощью менеджера пакетов pip. В отличие от обычной установки командой `pip3 install <имя_пакета>`, вместо имени пакета указывается путь к whl-файлу `pip3 install <путь/к/whl_файлу>`. Например:

```
pip3 install C:\temp\networkx-1.11-py2.py3-none-any.whl
```

Откройте окно командной строки и при помощи команды `cd` перейдите в каталог, где размещен ваш whl-файл. В Anaconda по умолчанию подобные файлы лежат в каталоге Scripts. Просто скопируйте туда ваш whl-файл. В этих случаях полный путь указывать не понадобится. Например:

```
pip3 install networkx-1.11-py2.py3-none-any.whl
```

При выборе пакета важно, чтобы разрядность устанавливаемой библиотеки и разрядность интерпретатора совпадали. Пользователи Windows могут брать whl-файлы с сайта http://www.lfd.uci.edu/~gohlke/pythonlibs/. Библиотеки там постоянно обновляются и в архиве содержатся все, какие только могут понадобиться.

Подготовка среды Python 3 в ОС Fedora 24

Действие	Команда
Обновление ОС Fedora 24 (http://losst.ru/nastrojka-fedora-24-posle-ustanovki)	`$ dnf update` `$ dnf upgrade`

Интерпретатор языка Python 3 и менеджер пакетов	`$ dnf install python3`
	`$ dnf -y install python3-tools`
	`$ pip3 install --upgrade pip`

Оболочка Python IDLE3	`$ dnf install idle3-tools`
Основные научные библиотеки	`$ dnf install python3-numpy python3-matplotlib python3-scipy python3-pandas python3-sympy python3-scikit-learn`
Записные книжки Jupyter	`$ pip3 install jupyter --upgrade`

Инструментальная среда Spyder (файл whl в каталоге пакетов Python https://pypi.python.org/pypi/spyder)	`$ dnf install python3-spyder`
	или
	`$ dnf install PyQt5`
	`(pip3 install PyQt5)`
	`$ sudo dnf install python3-devel`
	`$ pip3 install psutil`
	`$ sudo pip3 install /путь/к/файлу/spyder-3.0.0-py3-none-any.whl`

СУБД PostgreSQL, геопространственного расширения PostGIS, адаптера базы данных psycopg2 и администратора БД pgadmin3	`$ sudo dnf list postgresql\*`
	`$ sudo dnf install postgresql postgresql-contrib postgresql-server postgis`
	`$ sudo dnf install python3-psycopg2`
	`$ sudo dnf install pgadmin3`

Установка геопространственных библиотек в ОС Fedora 24

Библиотека GDAL

Скачать файл RPM из каталога библиотек RPM https://www.rpmfind.net/linux/rpm2html/search.php?query=gdal-python3:

```
$ dnf install /путь/к/файлу/gdal-python3-2.1.1-1.fc26.x86_64.rpm
```

или

```
$ dnf install gdal gdal-devel
```

Библиотека pyproj

Скачать файл RPM из каталога библиотек RPM https://www.rpmfind.net/linux/rpm2html/search.php?query=python3-pyproj&submit=Search+...&system=&arch=:

```
$ dnf install /путь/к/файлу/python3-pyproj-1.9.5.1-2.fc24.x86_64.rpm
```

Библиотека Shapely

Скачать файл RPM из каталога библиотек RPM https://www.rpmfind.net/linux/rpm2html/search.php?query=python3-shapely&submit=Search+...&system=&arch=:

```
$ dnf install /путь/к/файлу/python3-shapely-1.5.16-1.fc24.x86_64.rpm
```

Библиотека Mapnik

Автор книги будет использовать библиотеку Mapnik в главах 3, 7, 8, 9 и 13. Стоит отметить, что версия 3 библиотеки Mapnik, являясь мощным средством картографирования, пока работает только в Linux и Mac OS X, и разработка бинарников для Windows стоит в ближайших планах разработчиков. В связи с этим примеры программ в этих главах тестировались в ОС Fedora 24. Кроме того, предыдущие издания этой книги вышли в 2010 и 2013 гг., и с тех пор парадигма написания кода с использованием библиотеки Mapnik поменялась с процедурного на декларативный на основе XML и Geojson. Тем не менее в книге авторские примеры программного кода оставлены без изменений, а альтернативные реализации на XML приведены в комментариях и в прилагаемых к книге примерах.

Скачать файл RPM из каталога библиотек RPM https://www.rpmfind.net/linux/rpm2html/search.php?query=mapnik&submit=Search+...&system=&arch=:

```
$ dnf install /путь/к/файлу/mapnik-3.0.10-3.fc24.x86_64.rpm
$ dnf install /путь/к/файлу/python3-mapnik-0.1-7.20160202git1cb6851.fc24.x86_64.rpm
```

Запись справочной информации по библиотеке mapnik в HTML-файл:

```
$ python3
>>> import pydoc
>>> import mapnik
>>> pydoc.writedoc("mapnik")
>>> exit()
```

Протокол настройки СУБД PostgreSQL в ОС Fedora 24

```
[root@localhost ~]$ sudo postgresql-setup --initdb --unit postgresql
[root@localhost ~]$ sudo systemctl start postgresql
[root@localhost ~]$ sudo -i -u postgres
-bash-4.3$ psql
psql (9.5.4)
Введите "help", чтобы получить справку.
postgres=# alter user postgres password 'geo';
ALTER ROLE
postgres=# \c
Вы подключены к базе данных "postgres" как пользователь "postgres".
postgres=# create table test (id int);
CREATE TABLE
postgres=# \d
          Список отношений
 Схема  | Имя  |   Тип   | Владелец
--------+------+---------+----------
 public | test | таблица | postgres
(1 строка)
postgres=# drop table test;
DROP TABLE
postgres=# \d
```

```
Отношения не найдены.
postgres=# \q
-bash-4.3$ psql -c "GRANT ALL PRIVILEGES ON DATABASE postgres TO postgres"
GRANT
-bash-4.3$ psql -d postgres -c "CREATE EXTENSION postgis;"
CREATE EXTENSION
-bash-4.3$ exit
logout
[root@localhost ~]$ sudo systemctl stop postgresql
[root@localhost /]# nano /var/lib/pgsql/data/pg_hba.conf
  Прим. Отредактировать 2 строки в pg_hba.conf и сохранить файл:
  local all all peer  поменять на local all all trust
  host  all all 127.0.0.1/32 ident поменять на host  all all 127.0.0.1/32 md5
[root@localhost ~]$ sudo systemctl start postgresql
```

Протокол установки и настройки Django в ОС Fedora 24

```
[user@localhost /]$ sudo pip3 install django
[user@localhost /]$ cd путь/к/djex              # djex - это пример приложения
[user@localhost djex]$ django-admin startproject example
[user@localhost djex]$ ls
example
[user@localhost djex]$ cd example
[user@localhost example]$ python3 manage.py startapp hello
    (обновить settings.py)
    (обновить models.py)
[user@localhost example]$ python3 manage.py makemigrations hello
Migrations for 'hello':
  hello/migrations/0001_initial.py:
    - Create model Counter
[user@localhost example]$ python3 manage.py migrate
Operations to perform:
  Apply all migrations: auth, contenttypes, hello, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0001_initial... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying hello.0001_initial... OK
  Applying sessions.0001_initial... OK
    (обновить examples/urls.py)
    (обновить hello/views.py)
    (создать hello/templates/say_hello.html)
```

```
[user@localhost example]$ python3 manage.py runserver
Performing system checks...
System check identified no issues (0 silenced).
October 07, 2016 - 07:41:34
Django version 1.10.2, using settings 'example.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

(набрать в браузере http://127.0.0.1:8000)

Глава 1

Разработка геопрограмм на Python

Эта глава предлагает обзор языка программирования Python и процесса разработки геопространственных приложений. Отметим, что данная глава не является учебным руководством по использованию языка Python; язык Python прост в изучении, однако связанные с этим подробности выходят за рамки данной книги.

В этой главе будут рассмотрены следующие темы:

- что из себя представляет язык Python и чем он отличается от других языков программирования;
- каким образом стандартная библиотека Python и его каталог пакетов делают этот язык еще более мощным;
- что обозначают термины **геопространственные данные** и **разработка геопространственных приложений**;
- обзор процесса получения доступа к геопространственным данным, управления ими и их отображения на экране компьютера, и каким образом он осуществляется;
- несколько основных сфер приложения разработок в области геопространственного программирования;
- некоторые современные тенденции развития процесса разработки геопространственного программного обеспечения.

Python

Python (http://python.org) – это современный, высокоуровневый язык программирования, подходящий для широкого спектра задач программирования. Он часто используется в качестве языка сценариев для автоматизации и упрощения задач на уровне операционной системы и одинаково подходит для создания больших и сложных программ. Python использовался при написании веб-систем, настольных приложений, игр, научных программ и даже утилит и других высокоуровневых элементов различных операционных систем.

Python поддерживает широкий спектр идиом программирования от прямого процедурного программирования до объектно-ориентированного и функционального программирования.

Python иногда критикуют за то, что он – интерпретируемый и бывает медленным, по сравнению с компилирующими языками, такими как C. Однако использование байткодовой компиляции и тот факт, что подавляющая часть тяжелой работы выполняется программными библиотеками, означают, что производительность Python часто бывает удивительно хороша – к тому же имеется множество приемов, которые позволяют улучшить производительность программ, если это необходимо.

Версии интерпретатора Python с открытым исходным кодом находятся в свободном доступе для всех основных операционных систем. Python идеально подходит для всех видов программирования, от оперативных разовых сценариев до создания огромных и сложных систем. Python может даже выполняться в интерактивном режиме (в командной строке), позволяя вводить одноразовые команды и короткие программы и сразу получать результаты. Это идеальный сценарий для выполнения оперативных вычислений или выяснения того, как работает конкретная библиотека.

Первое, на что разработчик на Python обращает внимание, по сравнению с другими языками, такими как Java или C++, – это выразительность языка: то, что на Java потребует 20–30 строк программного кода, на Python часто занимает менее 10 строк. Например, предположим, необходимо напечатать сортированный список слов, которые встречаются в конкретной части текста. В Python это достигается легко:

```
words = set(text.split())
for word in sorted(words):
    print(word)
```

Реализация этого вида задачи на других языках часто, к удивлению, представляет трудности.

В то время как непосредственно сам язык Python позволяет программировать быстро и легко, давая возможность фокусироваться на поставленной задаче, стандартная библиотека Python делает программирование еще более эффективным. Эта библиотека облегчает выполнение таких процедур, как конвертирование значений даты и времени, обработка строковых значений, загрузка данных с вебсайтов, выполнение сложных математических вычислений, работа с электронными письмами, кодирование и декодирование данных, разбор XML, шифрование данных, файловые операции, сжатие и распаковка файлов, работа с базами данных – и далее по списку. То, что можно сделать с использованием стандартной библиотеки Python, воистину впечатляет.

Наряду со встроенными в стандартную библиотеку Python модулями можно легко загрузить и установить пользовательские модули, которые могут быть написаны на Python или C. Каталог библиотек языка Python (http://pypi.python.org) предлагает тысячи дополнительных модулей, которые можно загрузить

и установить. И если этого недостаточно, то множество других систем предлагает привязки Python[1], которые обеспечивают к ним доступ непосредственно из ваших программ. Кстати, в этой книге мы будем интенсивно использовать питоновские привязки.

Python – это во многих отношениях идеальный язык программирования. Познакомившись с языком и пару раз использовав его в деле, вы осознаете, насколько он легок при написании программ для решения разнообразных задач. Не опасаясь похоронить себя в болоте определений типов и среди низкоуровневых методов обработки строк, можно просто сконцентрироваться на том, чего вы хотите достигнуть. И в итоге придете к тому, что начнете думать непосредственно в терминах питоновского программного кода. Программирование на Python прямолинейное, эффективное и, не побоюсь этого слова, увлекательное.

Python 3

Сегодня используются две главные разновидности Python: Python версии 2.x существует уже в течение многих лет и все еще широко используется, в то время как Python версии 3.x, которая не является обратно несовместимой с Python 2, становится все более популярным, учитывая, что эта версия Python взята за основу для дальнейшего развития.

Один из главных факторов, сдерживающих повсеместное принятие Python 3, – это отсутствие поддержки сторонних библиотек. Эта проблема было особенно острой относительно библиотек Python, использующихся для разработки геопространственных приложений, поскольку они часто зависят от отдельных разработчиков или имеют требования, которые не были совместимыми с Python 3 в течение достаточно продолжительного времени. Однако все основные библиотеки, которые используются в этой книге, теперь в равной степени можно выполнять, используя Python 3, и поэтому все примеры программного кода в этой книге преобразованы таким образом, чтобы использовать синтаксис этой версии Python.

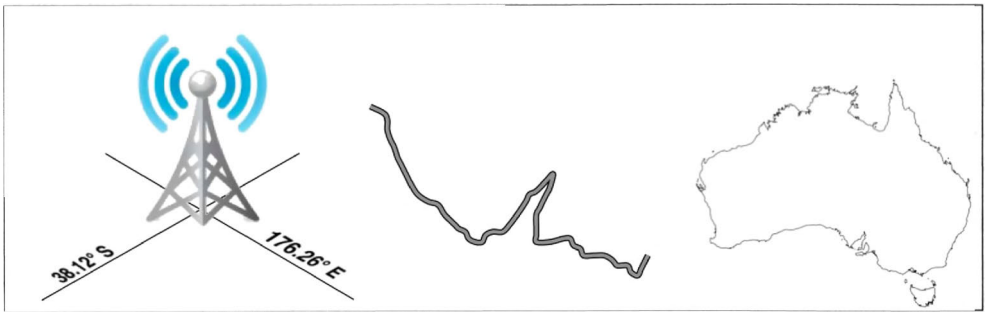
Если ваш компьютер работает под Linux или Mac OS X, то вы можете непосредственно использовать Python 3 со всеми этими библиотеками. Если же ваш компьютер работает под Windows, то совместимость с Python 3 выглядит проблематичнее. В этом случае имеются две возможности: можно попытаться скомпилировать библиотеки самостоятельно, чтобы получить возможность работать с Python 3, либо вернуться к использованию Python 2 и внести изменения в примеры программного кода, где это требуется. К счастью, различия в синтаксисе между Python 2 и Python 3 предельно простые и поэтому потребуют незначительных изменений, если вы действительно примете решение использовать Python 2.x, а не Python 3.x.

---

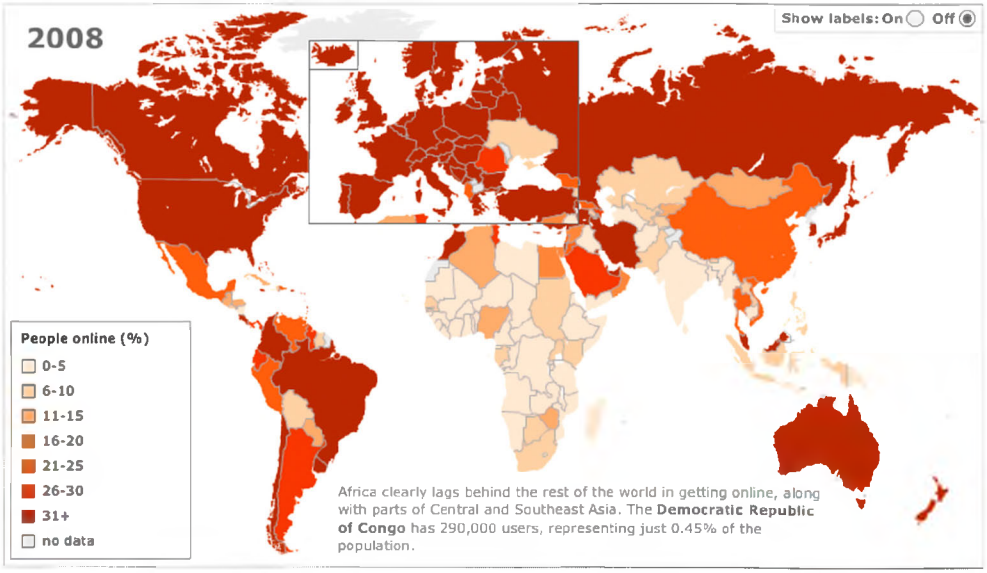
[1] Привязка (binding) – программный код, благодаря которому программа получает доступ к методам и свойствам программного объекта. В сущности, это оберточный или промежуточный слой между средой разработки на Python и прикладным программным интерфейсом (API), который предоставляется веб-сервисом, динамической библиотекой или СУБД. – *Прим. перев.*

Разработка геопространственных программ

Термин **«геопространственный»** относится к отысканию информации, которая расположена на земной поверхности. Например, она может состоять из географического положения вышек сотовой связи, географической формы дороги или контура страны:

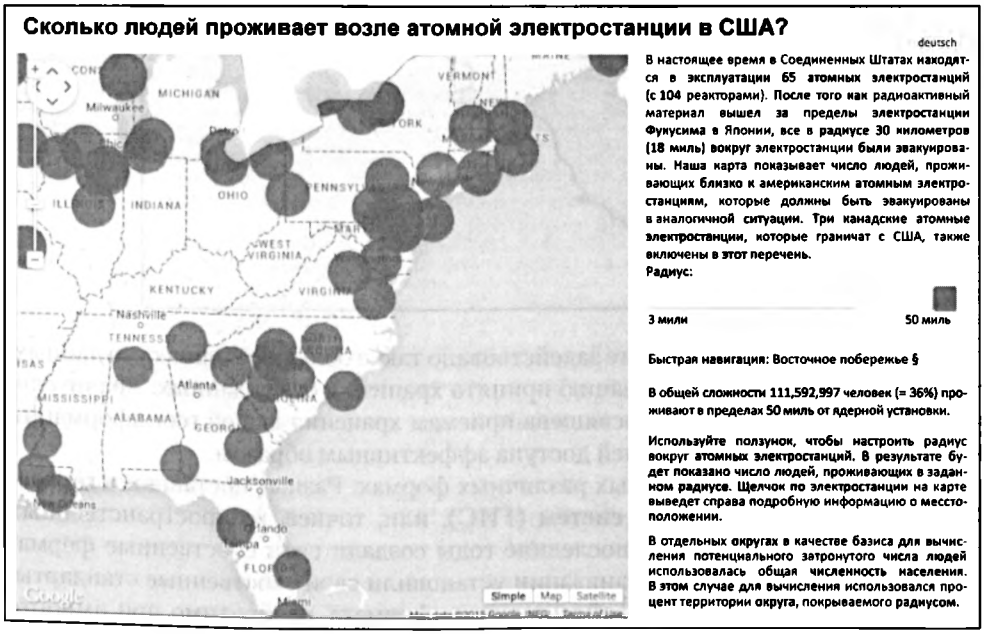


Геопространственные данные часто связывают некую порцию информации с конкретным географическим положением. Например, ниже приведена интерактивная карта с веб-сайта http://www.bbc.co.uk, которая показывает процент населения в каждой стране с доступом в Интернет в 2008 г.



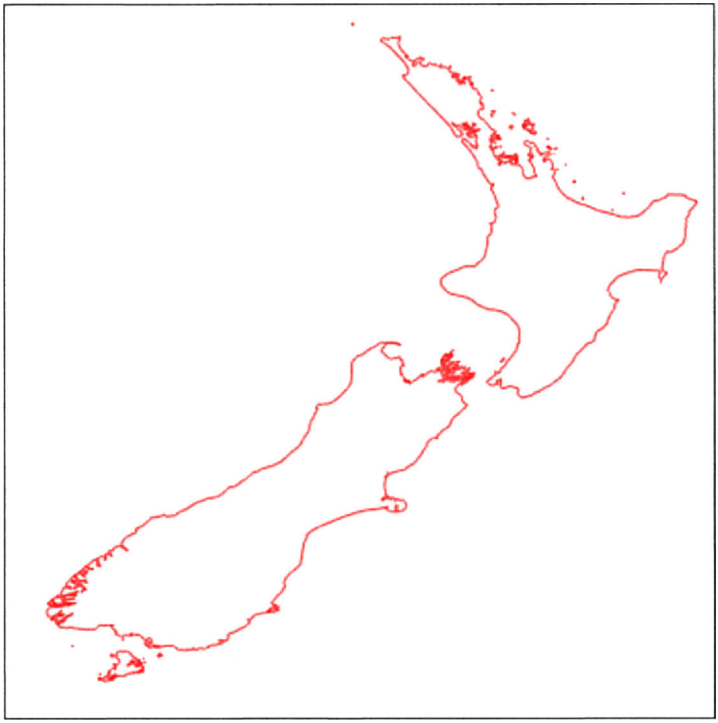
Аналогичная карта из Википедии с данными за 2012 г. (https://en.wikipedia.org/wiki/Global_Internet_usage):

Геопространственные данные, или просто *геоданные*, как правило, связывают некоторое количество информации с конкретным географическим положением. Например, следующая ниже карта, взятая с http://opendata.zeit.de/nuclear-reactors-usa, показывает число людей, проживающих в пределах 50 миль от ядерного реактора в границах восточной части США:



Термин «**разработка геопространственных приложений**» обозначает процесс написания компьютерных программ, которые способны получать доступ к такому типу информации, управлять ею и ее визуализировать.

Внутренне геоданные представлены в виде серии **координат**, часто в виде значений широты и долготы. Кроме того, нередко также присутствуют дополнительные **атрибуты**, такие как температура, тип почвы, высота или название ориентира. Одиночный набор геоданных может описывать до нескольких тысяч (или даже миллионов) точек данных. Например, следующий ниже контур Новой Зеландии состоит почти из 12 000 отдельных точек данных:



По причине того, что в работе задействовано такое большое количество данных, геопространственную информацию принято хранить в базах данных. Значительная часть этой книги будет посвящена приемам хранения вашей геоинформации в базе данных и получения к ней доступа эффективным образом.

Геоданные поступают в самых различных формах. Разные поставщики **географических информационных систем** (ГИС), или, точнее, геопространственных информационных систем, за последние годы создали свои собственные форматы файлов, а разного рода организации установили свои собственные стандарты. И поэтому, чтобы прочитать файлы нужного формата, необходимо при импорте геоданных в вашу базу данных пользоваться библиотекой Python.

К сожалению, не все точки геоданных совместимы. Точно так же, как расстояние величиной 2.8 единицы может иметь совсем разные значения в зависимости от того, используете вы километры или мили, конкретное значение координаты может указывать на любое число точек на искривленной поверхности Земли, в зависимости от того, какая **картографическая проекция** используется.

Картографическая проекция – это метод представления земной поверхности в двух измерениях. Мы рассмотрим проекции более подробно в *главе 2 «Геоинформационные системы»*, но на данный момент просто следует иметь в виду, что с любым элементом геоданных связана проекция. Чтобы сравнить или совместить два набора геоданных, часто необходимо конвертировать данные из одной проекции в другую.

Значения широты и долготы иногда упоминаются как неспроецированные, или географические, координаты. Мы узнаем подробнее об этом в следующей главе.

В дополнение к прозаическим задачам импорта геоданных из различных внешних форматов файлов и перевода данных из одной проекции в другую геоданными можно управлять, чтобы решать разного рода содержательные задачи. Очевидные примеры включают в себя задачу вычисления расстояния между двумя точками, задачу вычисления длины дороги или нахождения всех точек данных, находящихся в пределах заданного радиуса от выбранной точки. Чтобы решать все эти и другие задачи, мы будем пользоваться библиотеками Python.

И наконец, сами по себе геоданные не очень интересны. Длинный список координат практически ни о чем не говорит; понять их смысл можно, только когда эти числа используются для создания изображений. Составление карт, размещение на них точек данных и предоставление пользователям возможности с ними взаимодействовать – все эти задачи являются важными аспектами процесса разработки геоприложений.

Мы рассмотрим их все в более поздних главах.

Сферы применения геоприложений

Вкратце рассмотрим некоторые наиболее распространенные задачи, решаемые геоприложениями, с которыми вы, по всей видимости, уже встречались.

Анализ геоданных

Предположим, у вас есть база данных, которая содержит набор геоданных по г. Сан-Франциско. Эта база данных может состоять из географических объектов, дорожной сети, географических положений видных строений и других антропогенных объектов, таких как мосты, аэропорты и т. д.

Такая база данных может быть ценным ресурсом для ответа на различные вопросы, такие как:

- Какая самая длинная дорога в г. Саусалито?

- Сколько мостов находится в г. Окленд?
- Какова общая площадь городского парка «Золотые ворота» (Golden Gate Park) в г. Сан-Франциско?
- Каково расстояние от Пирса 39 до Башни Койт[1]?

Многие из этих типов задач можно решить при помощи таких инструментов, как расширение PostGIS со встроенной поддержкой пространственных данных. Например, чтобы рассчитать общую площадь городского парка «Золотые ворота», можно воспользоваться следующим запросом SQL:

```
select ST_Area(geometry) from features
where name = "Golden Gate Park";
```

Чтобы рассчитать расстояние между двумя географическими положениями, их сначала необходимо геокодировать, получив их значения широты и долготы. Это можно выполнить разными методами; один из самых простых состоит в использовании бесплатной веб-службы геокодирования, как, например, эта:

```
http://nominatim.openstreetmap.org/search?format=json&q=Pier 39,San Francisco, CA
```

Для Пирса 39 она возвращает (среди прочего) значения широты `37.8101274` и долготы `-122.4104622`.

Приведенные значения широты и долготы исчисляются десятичными градусами. Если вы не знаете, что это такое, то не переживайте; мы займемся ими в главе 2 «Геоинформационные системы».

Аналогичным образом можно установить географичесое положение Башни Койт, используя для этого следующий ниже запрос:

```
http://nominatim.openstreetmap.org/search?format=json&q=Coit Tower, San Francisco, CA
```

Он возвращает значение широты `37.80237485` и долготы `-122.405832766082`.

Располагая координатами двух нужных географических точек, мы можем рассчитать между ними расстояние, используя для этих целей библиотеку Python `pyproj`:

Если вы захотите выполнить этот пример, то сначала необходимо инсталлировать библиотеку `pyproj`. Мы рассмотрим инсталляцию библиотек в *главе 3 «Библиотеки Python для геопрограммирования»*.

```
import pyproj

lat1,long1 = (37.8101274,-122.4104622)
lat2,long2 = (37.80237485,-122.405832766082)

geod = pyproj.Geod(ellps="WGS84")
```

---

[1] Башня Койт (Coit Tower) – башня-мемориал, расположенная в парке пионеров (Pioneer Park) на вершине Телеграфного холма (Telegraph Hill). Со смотровой площадки башни открывается уникальный вид на Сан-Франциско. Благодаря этому факту башня является излюбленным местом посещения туристами. – *По материалам Википедии.*

```
angle1,angle2,distance = geod.inv(long1, lat1, long2, lat2)
print("Расстояние составляет {:0.2f} метров".format(distance))
```

В результате будет выведено расстояние между двумя точками:

Расстояние составляет 952.17 метра.

На данном этапе вам не стоит обращать внимания на обозначение `WGS84`; его смысл будет рассмотрен в *главе 2 «Геоинформационные системы»*.

Разумеется, в обычных условиях такого рода анализ не выполняется на разовой основе, как тут, – гораздо чаще на Python создают программу, которая отвечает на подобные вопросы для *любого* нужного набора данных. Можно, к примеру, создать веб-приложение, которое выводит на экран меню доступных численных расчетов. Одна из опций в этом меню могла бы выполнять расчеты расстояния между двумя точками; при выборе этой опции веб-приложение предложит пользователю ввести географические положения этих двух точек, попытается их геокодировать, вызвав для этого соответствующую веб-службу (и выведет на экран компьютера сообщение об ошибке, если геокодировать географическое положение не получилось), затем рассчитает между двумя точками расстояние, используя библиотеку `pyproj`, и, наконец, выведет пользователю результаты.

С другой стороны, если у вас есть база данных, которая содержит соответствующие геоданные, то можно предоставить пользователю возможность выбрать географические положения этих двух точек из базы данных, вместо того чтобы заставлять пользователя в произвольном режиме набирать наименования географических положений или уличные адреса на клавиатуре.

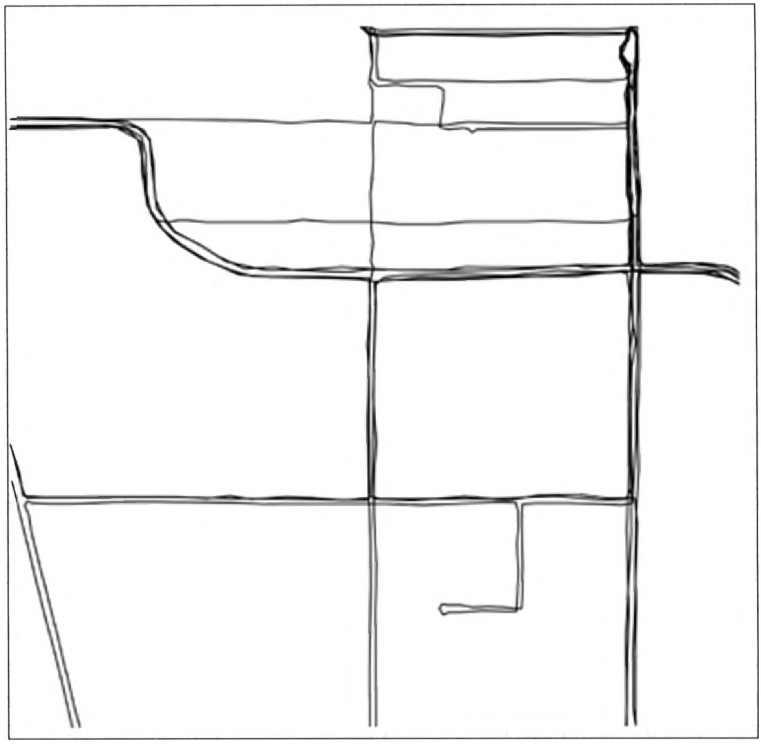
Каким бы ни было ваше решение о том, как структурировать ввод данных, выполнение подобного рода расчетов будет главной составной частью вашего геопространственного приложения.

Визуализация геоданных

Предположим, вам надо посмотреть, какие районы города обычно покрываются услугами такси в течение среднего рабочего дня. В автомобиль такси можно установить записывающее устройство GPS и оставить его там на несколько дней, чтобы оно записывало его географические положения. В результате получится серия временных меток, а также значения широт и долгот, как показано ниже:

```
03-21 9:15:23  -38.16614499  176.2336626
03-21 9:15:27  -38.16608632  176.2335635
03-21 9:15:34  -38.16604198  176.2334771
03-21 9:15:39  -38.16601507  176.2333958
...
```

Сами по себе эти голые цифры почти ничего не означают. Но когда вы визуализируете эти данные на экране компьютера, цифры начинают приобретать смысл:



Процедура скачивания примеров программного кода подробно описана в *предисловии* к этой книге.

Пакет примеров программного кода, прилагаемый к данной книге, размещен на странице GitHub https://github.com/PacktPublishing/Python-Geospatial-Development-Third-Edition. Мы также располагаем другими пакетами примеров, которые можно выбрать из нашего богатого каталога книг и видео, предлагаемого на странице https://github.com/PacktPublishing/. Можете убедиться сами!

Можно сразу увидеть, что такси имеет тенденцию снова и снова перемещаться вдоль одних и тех же улиц, и если вы наложите эти данные поверх карты города, то увидите, где именно такси находилось:

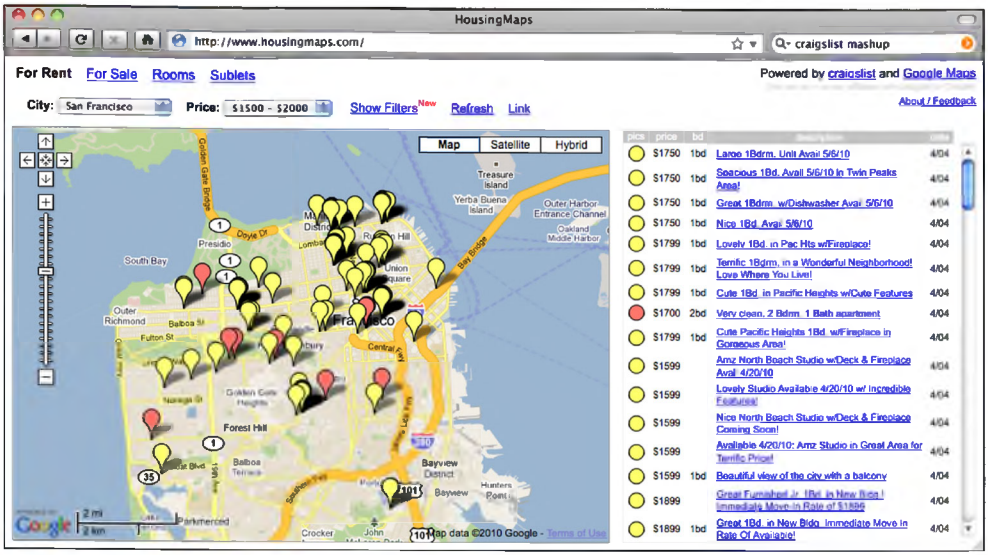
Карта города любезно предоставлена http://openstreetmap.org

Хоть это и очень простой пример, визуализация является решающим аспектом работы с геоданными. Методы визуального отображения данных на экране компьютера, наложения разных наборов данных и прямого визуального управления данными пользователем будут главными темами в этой книге.

Создание геопространственных мэшапов

В последние годы понятие *мэшапа* (mash-up) стало в информатике популярным. Мэшапы – это гибридные приложения, которые совмещают данные и функциональность из нескольких источников.

Например, типичный мэшап состоит из выборки описаний домов, сдаваемых в аренду в конкретном городе, и визуализации географических положений арендных плат на карте, как показано ниже:



Изображение любезно предоставлено http://housingmaps.com

При создании подобного рода мэшапов до сих пор широко используется прикладной программный интерфейс картографической веб-службы Карты Google. Однако эта картографическая веб-служба имеет некоторые ограничения, в том числе связанные с лицензированием. Впрочем, она не является единственной, и такие картографические инструменты, как динамическая библиотека Mapnik, библиотека OpenLayers на JavaScript и платформа MapServer, среди многих других тоже позволяют создавать мэшапы, причем с наложением на карту своих собственных данных.

Большинство таких мэшапов работает как веб-приложения в Интернете, выполняясь на сервере, к которым может получить доступ любой, у кого есть веб-браузер. Иногда мэшапы имеют конфиденциальный характер, требуя доступа по паролю, но обычно они бесплатны и могут использоваться любым, кто пожелает. Действительно, много компаний (таких как веб-сайт карт жилых комплексов `housingmaps.com`, показанный на предыдущем снимке) основано на общедоступных геопространственных мэшапах.

Последние достижения

Десятилетие назад возможности разработки геоприложений были в значительной степени более ограничены, чем сегодня. Нормой для работы и визуализации геоданных были профессиональные (и чрезвычайно дорогие) географические информационные системы. Инструментальные средства с открытым исходным кодом там, где они были доступны, были непонятны и сложны в использовании. Более того, все работало исключительно на настольном компьютере – понятие работы с геоданными в Интернете было не более, чем отдаленной мечтой.

В 2005 г. компания Google выпустила два продукта, которые полностью изменили характер геопрограммирования: картографическая веб-служба Карты Google и бесплатная программа просмотра карт на основе трехмерной модели земного шара Google Планета Земля позволили любому с веб-браузером или настольным компьютером просматривать геоданные и с ними работать. Не требуя экспертных знаний и лет наработки опыта, они позволили даже четырехлетнему ребенку непосредственно просматривать и управлять интерактивными картами мира.

Продукты Google не совершенны: картографические проекции сознательно упрощены, приводят к ошибкам и проблемам при отображении на экране наложений. Эти продукты общедоступны исключительно для некоммерческого использования и почти не содержат функционала, связанного с выполнением геоанализа. Несмотря на эти ограничения, они оказали на процесс разработки геоприложений огромное воздействие. Люди узнали об открывшихся возможностях при использовании карт, а лежащие в их основе геоданные стали настолько распространены, что теперь обычно включают встроенные картографические инструменты даже в сотовые телефоны.

Система глобального позиционирования, англ. термин Global Positioning System (GPS), также оказала определяющее влияние на процесс разработки геоприложений. Геоданные для улиц и других антропогенных и природных объектов раньше были дорогим и жестко регламентированным ресурсом, который часто создавался путем сканирования аэрофотоснимка и затем ручной прорисовки контуров улиц или береговой линии поверх скана, с тем чтобы потом оцифровать необходимые участки. С появлением дешевых и легко доступных портативных модулей GPS, а также телефонов, куда GPS уже встроен, теперь любой по желанию может получать свои собственные геоданные. Действительно, у многих людей стало хобби записывать, редактировать и улучшать точность уличных и топологических данных, которые затем свободно поступают в совместное пользование через Интернет. Все это означает, что вы не ограничиваетесь записью ваших собственных данных или покупкой данных у коммерческой организации; добровольно предложенная информация теперь часто так же точна и практична, как и коммерчески доступные данные, и может с успехом подойти для вашего геоприложения.

Кроме того, свой вклад в процесс разработки геоприложений внесли успехи движения за программное обеспечение с открытым исходным кодом. Вместо того чтобы полагаться на коммерческие комплекты инструментов, теперь появилась возможность создавать сложные геоприложения, полностью состоящие из свободных инструментов и программных библиотек. Поскольку исходный код этих инструментов часто находится в общем доступе, разработчики могут их улучшать и расширять, решая возникшие проблемы и добавляя новый функционал на пользу всем. Такие инструменты, как динамическая библиотека PROJ.4, геопространственное расширение PostGIS, пакет программных библиотек GDAL/OGR, представляют собой превосходные инструменты для геопространственной обработки данных, которые появились благодаря движению за ПО с открытым исходным кодом. Мы будем использовать все эти инструменты на протяжении всей книги.

Наряду с автономными инструментами и библиотеками появилось много геопространственных **прикладных программных интерфейсов**, англ. термин Application Programming Interface (API). Компания Google предложила ряд программных интерфейсов, которые могут использоваться для внедрения карт и выполнения ограниченного геоанализа на веб-сайте. Другие сайты, такие как геокодер OpenStreetMap, который мы использовали ранее, позволяют выполнять различные геопространственные задачи, которые было бы трудно решить, если бы вы были ограничены использованием ваших собственных данных и программных ресурсов.

По мере того как от растущего числа источников поступает все больше и больше геоданных, и поскольку одновременно с этим увеличивается число инструментов и систем, которые могут работать с этими данными, все насущней становится задача установления для геопространственных данных соответствующих **стандартов**. Международная организация по стандартизации Открытый геопространственный консорциум, англ. термин Open Geospatial Consortium (OGC) (http://www.opengeospatial.org), стремится заниматься именно тем, что внедряет ряд стандартных форматов и протоколов совместного использования и хранения геоданных. Эти стандарты, включая GML, KML, GeoRSS, WMS, WFS и WCS, предлагают универсальный язык, на котором можно описывать геоданные. Такие инструменты, как коммерческие и открытые геоинформационные системы, бесплатная программа для просмотра карт Google Планета Земля, разнообразные веб-API и специализированные геопространственные библиотеки наподобие динамической библиотеки OGR, – все они в состоянии работать с этими стандартами. По-настоящему важный аспект инструментов геообработки состоит в их способности распознавать данные и транслировать их между этими разными форматами.

По мере того как устройства со встроенными GPS-приемниками становятся все более повсеместными, стало возможным выполнять запись данных о вашем местоположении, выполняя при этом другую задачу. **Геолокация**, то есть определение и/или запись вашего местоположения, в то время как вы делаете что-то еще, получает все большее распространение. Социальная сеть Twitter, например, теперь позволяет делать запись и отображать свое текущее местоположение, когда вы публикуете обновление статуса. По мере приближения к своему офису технологически сложное программное обеспечение, отслеживающее список текущих дел, теперь автоматически может скрыть те задачи, которые не могут быть выполнены в данном месте. Кроме того, ваш телефон может напомнить вам, кто из ваших друзей находится неподалеку, а результаты поиска могут быть отфильтрованы так, чтобы показывать только близлежащие предприятия.

Все это является простым и логическим продолжением тенденции, которая стартовала, когда геоинформационные системы размещались на мэйнфреймовых компьютерах и управлялись специалистами, которые потратили годы на их изучение. В последние годы геоданные и геоприложения были «демократизированы», став доступными в большем количестве мест и большему количеству людей. То, что было возможно только в крупной организации, теперь может быть выполнено

любым пользователем карманного устройства. На фоне неуклонного улучшения технологии и увеличения мощности инструментов эта тенденция, несомненно, продолжится и в будущем.

Заключение

В этой главе мы вкратце представили язык программирования Python и базовые понятия, которые лежат в основе процесса разработки геопространственных приложений. Мы убедились, что Python – это очень высокоуровневый язык и что доступность сторонних библиотек для работы с геоданными делает его в высшей степени подходящим для задачи разработки геоприложений. Мы узнали, что термин «геопространственные данные» (или *геоданные*) обозначает отыскание расположенной на земной поверхности информации при помощи координат, а термин «разработка геопространственных приложений» обозначает процесс написания компьютерных программ, которые могут получать доступ к геоданным, управлять ими и их визуализировать.

Далее мы рассмотрели типы вопросов, на которые можно ответить в результате анализа геоданных, увидели, каким образом геоданные могут использоваться для визуализации, и узнали о геопространственных мэшапах, которые практичным и содержательным образом совмещают данные (обычно это геоданные).

Затем мы узнали, каким образом картографическая веб-служба Карты Google, бесплатная программа просмотра карт Google Планета Земля и появление дешевых и портативных модулей GPS «демократизировали» процесс разработки геоприложений. Мы увидели, как движение за программное обеспечение с открытым исходным кодом привело к появлению большого числа высококачественных общедоступных программных инструментов, предназначенных для разработки геоприложений, и узнали, что различные организации, занимающиеся разработкой стандартов, внедряют форматы и протоколы, для того чтобы геоданные можно было совместно использовать и хранить.

Наконец, мы познакомились с геолокацией и узнали, каким образом она используется с целью получения геоданных и работы с ними самым неожиданным и полезным образом.

В следующей главе мы более подробно рассмотрим традиционные географические информационные системы, в том числе многие важные понятия, которые необходимо знать для того, чтобы работать с геоданными. Кроме того, будут исследованы различные геопространственные форматы. В заключение мы воспользуемся языком Python для выполнения разнообразных вычислений с использованием геоданных.

Глава 2

Геоинформационные системы

Аббревиатура **ГИС** обозначает **географические информационные системы**, англ. термин Geographic Information Systems (GIS). Обычно под ними подразумеваются сложные вычислительные системы, связанные с хранением геоданных, управлением ими и их визуализацией. Кроме того, эта аббревиатура может также использоваться для обозначения более общего понятия – геоинформатики, то есть науки, на базе которой применяются системы ГИС.

В этой главе мы изучим:

- центральные понятия ГИС, которые потребуются в будущем: географическое положение, расстояние, единицы измерения, картографические проекции, геодезические датумы, системы координат и географические фигуры;
- несколько основных форматов данных, с которыми вы, скорее всего, встретитесь во время работы с геоданными;
- несколько процессов, связанных с работой непосредственно с геоданными.

Ключевые понятия ГИС

Работа с геоданными осложняется тем, что вы имеете дело с математическими моделями земной поверхности. Во многих случаях легко представить Землю как сферу, на которой можно разместить данные. Это было бы просто, если б не одно «но»: результаты будут неточными – Земля больше походит на приплюснутый сфероид, чем на идеальную сферу. Вот это отличие, а также ряд других математических сложностей, в которые мы не будем тут погружаться, и придают процессу представления точек, линий и площадей на земной поверхности довольно сложный характер.

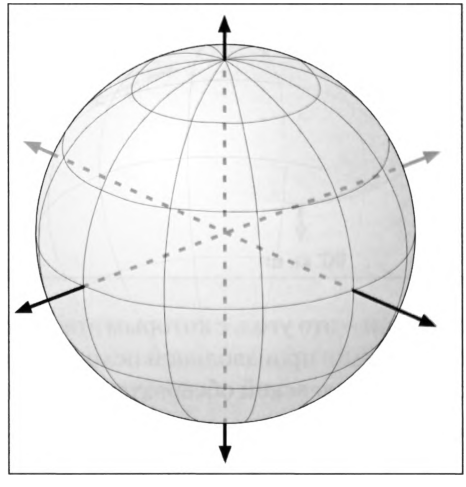
Рассмотрим некоторые ключевые понятия ГИС, которые необходимо знать при работе с геоданными.

Географическое положение

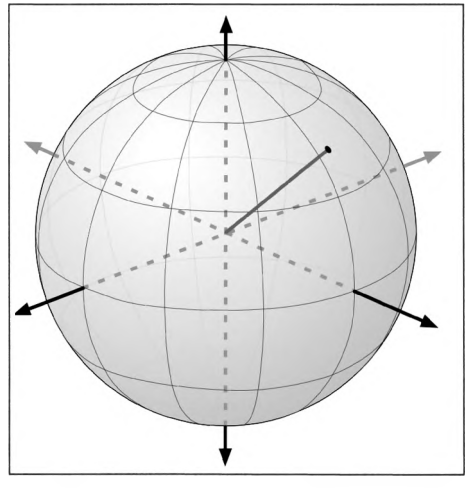
Географическое положение, или местоположение, – это точка на земной поверхности. Один из наиболее распространенных методов измерения географического

положения предполагает использование координат, определяемых двумя числами: широтой и долготой. Например, в настоящий момент мое географическое положение (согласно измерениям GPS-приемником) составляет 38,167446 градуса южной широты и 176,234436 градуса восточной долготы. Что эти числа означают и в чем их польза?

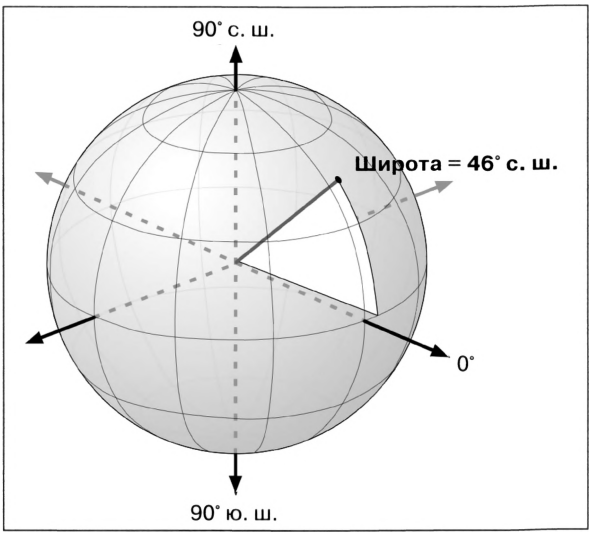
Представим Землю как полую сферу, у которой линии осей X, Y и Z проведены через ее центр:



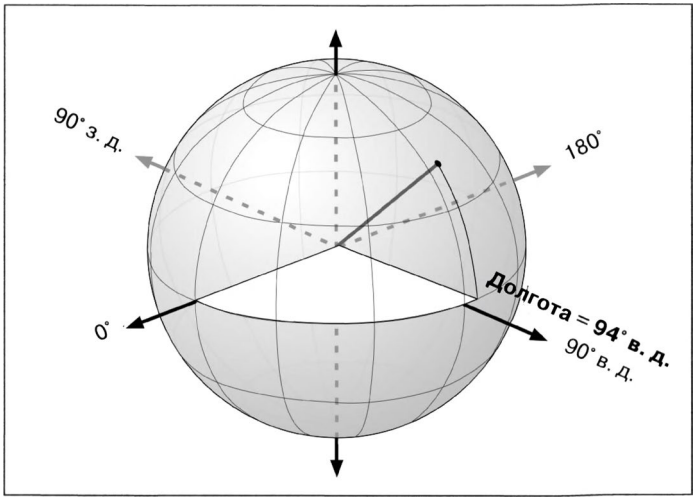
Любую заданную точку на земной поверхности можно соединить линией, исходящей из центра Земли. Это делается следующим образом:



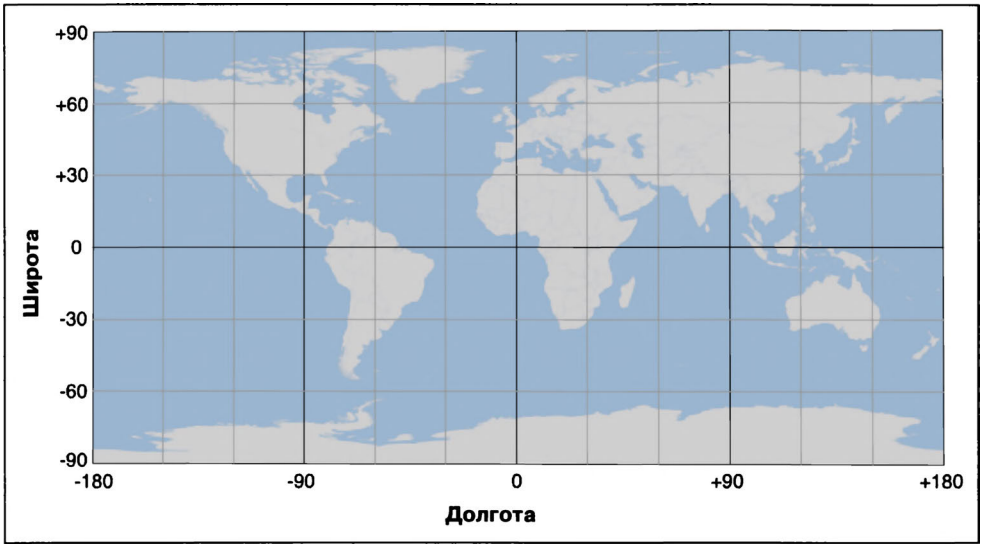
Широта точки – это угол, с которым линия проходит в направлении север–юг относительно плоскости экватора:



В то же время **долгота** точки – это угол, с которым эта линия проходит в направлении восток–запад относительно произвольной исходной точки (обычно берется географическое положение Королевской обсерватории в Гринвиче, Англия):



Принято, что положительные значения широты находятся в северном полушарии, тогда как ее отрицательные значения – в южном полушарии. Точно так же положительные значения долготы лежат восточнее Гринвича, и отрицательные значения – западнее от него. Таким образом, широты и долготы покрывают всю Землю, как показано ниже:



Горизонтальные линии, представляющие точки равной широты, называются **параллелями**, а вертикальные линии, представляющие точки равной долготы, называются **меридианами**. Меридиан в нулевой долготе часто называют **главным меридианом**. По определению параллель в нулевой широте – это экватор Земли.

Работая со значениями широты и долготы, следует помнить о двух моментах:

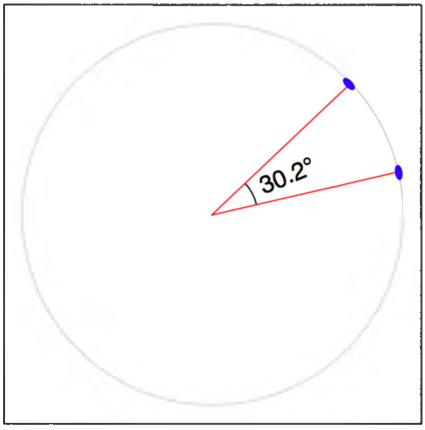
- западные долготы в основном отрицательные, но бывают ситуации (в особенности это касается данных, которые ограничены территорией США), когда западные долготы даны как положительные значения;
- значения долготы меняют знак и направление в точке ±180°. А именно, путешествуя на восток, долгота будет 177, 178, 179, 180, –179, –178, –177 и т. д., что может вконец запутать базовые вычисления расстояния, если полагаться только на свои силы, а не на программную библиотеку, которая может сделать эту работу за вас.

Значение широты и долготы относится к так называемому **геодезическому положению**. Геодезическое положение идентифицирует расчетную точку на поверхности Земли, независимо от того, что может находиться в этом месте. Значительная часть данных, с которыми мы будем работать, содержит геодезические положения. Тем не менее можно встретить и другие методы описания местоположения на поверхности Земли. Например, **место проживания физического лица** – это, понятно, просто уличный адрес, который представляет собой еще один совершенно допустимый (хоть и менее точный, с научной точки зрения) способ определения географического положения. Аналогичным образом **местонахождение юридического лица** содержит информацию о том, в пределах каких административно-территориальных границ (таких как избирательный округ, регион или город) оно находится, так как в некоторых контекстах эта информация представляет важность.

Расстояние

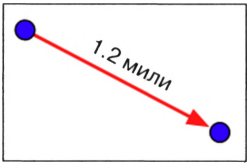
Расстояние между двумя точками на земной поверхности можно представить по-разному. Вот несколько примеров:

- **Угловое расстояние** – это угол между двумя лучами, исходящими из центра Земли через две точки.



Угловые расстояния обычно используются в сейсмологии; с ними можно встретиться во время работы с геоданными.

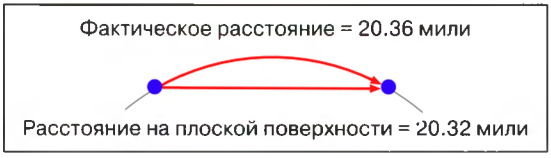
- **Линейное расстояние** – это именно то, что люди обычно подразумевают, когда говорят о расстоянии, – насколько далеко две точки отстоят друг от друга на земной поверхности.



Его часто называют расстоянием «по прямой». Вскоре мы обсудим это понятие более подробно, хотя следует уяснить, что линейные расстояния не так просты, как кажется.

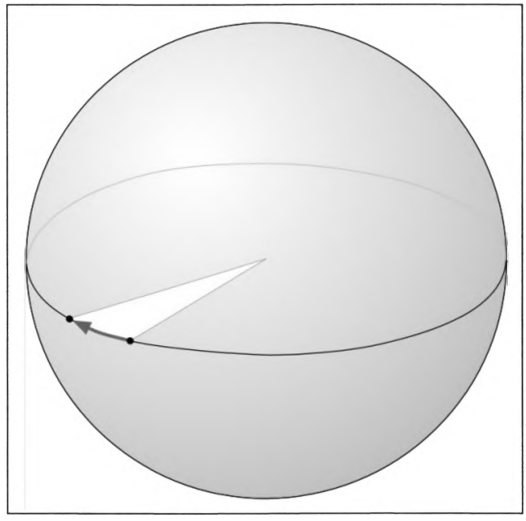
- **Пройденное расстояние**, или расстояние хода (traveling distance) – с линейными расстояниями («по прямой») все понятно, но не очень-то много наберется людей, которые умеют летать, как птицы. Другой полезный метод измерения расстояния состоит в измерении того, как далеко нужно фактически пропутешествовать, чтобы добраться от одной точки до другой, при этом обычно двигаясь по дороге или другому очевидному маршруту.

Подавляющую часть времени вы будете иметь дело с линейными расстояниями. Если бы Земля была плоской, то рассчитать линейные расстояния было бы легко: вы просто измеряете длину линии, проведенной между двумя точками, и все. Разумеется, Земля не плоская, что означает, что реальные расчеты расстояния, скорее, будут сложнее:



Так как мы работаем с расстояниями между точками, расположенными на земной, то есть не плоской, поверхности, мы на самом деле используем так называемое **расстояние по дуге большого круга** на сфере, англ. термин great-circle distance, у которого в геодезии имеется более строгий синоним – расстояние по ортодромии[1]. Расстояние по дуге большого круга – это длина полукруга между двумя точками на поверхности Земли, где полукруг центрирован вокруг середины Земли:

[1] Ортодромия (из др.-греч. «прямой путь») в геометрии – кратчайшая линия между двумя точками на поверхности вращения, частный случай геодезической линии. – *По материалам Википедии.*



Если вы предполагаете, что Земля сферическая, то вычислить расстояние по дуге большого круга между любыми двумя точками относительно просто; для этого часто используется **формула гаверсинуса**. Имеется и более сложная методика, которая позволяет более точно представить фигуру Земли[1], впрочем, формулы гаверсинуса будет достаточно для большинства случаев.

Далее в этой главе мы научимся использовать формулу гаверсинуса.

Единицы измерения

В сентябре 1999 г. орбитальный модуль Mars Climate Orbiter достиг внешних краев марсианской атмосферы, пропутешествовав в космосе в течение 286 дней. Создание аппарата обошлось в общей сложности в 327 миллионов долларов. Когда он приблизился к своей заключительной орбите, погрешность в расчете заставила его лететь слишком низко, и в результате орбитальный модуль был утерян. Расследование показало, что акселераторы аппарата вычисляли ускорение, используя английские единицы измерения, в то время как компьютер космического аппарата работал с метрическими единицами. Результатом явились катастрофа для НАСА и жесткое напоминание всем нам о важности понимания того, в каких единицах находятся ваши данные.

Геопространственные значения могут измеряться в самых разных единицах. Разумеется, расстояния могут измеряться в единицах метрической и английской системы мер, однако конкретное расстояние может на самом деле измеряться самыми разными единицами, в том числе используя следующие:

---

[1] См. статью, раскрывающую понятие *фигуры Земли* в Википедии: https://ru.wikipedia.org/wiki/Фигура_Земли, а также курс лекций по теории фигуры Земли: http://lnfm1.sai.msu.ru/grav/russian/lecture/tfe/index.html. – *Прим. перев.*

- дюйм;
- сантиметр;
- миллиметр;
- международный фут;
- геодезическая миля США;
- миля;
- метр;
- ярд;
- километр;
- международная морская миля;
- статутная миля США;
- британская морская миля.

Работая с данными, которые описывают расстояния, всегда важно знать, в каких единицах они измеряются. Более того, вам часто придется конвертировать данные из одной единицы измерения в другую.

Угловые меры тоже могут находиться в разных единицах: в градусах или радианах[1], и, значит, опять же вам часто придется конвертировать эти единицы из одной в другую.

Если подходить строго, то единицы измерения широты и долготы идентичны, и тем не менее вам часто придется иметь дело с различными форматами отображения их значений. Традиционно значения долготы и широты записываются в формате «градусы/минуты/секунды», как показано ниже:

```
176° 14' 4"
```

Еще один возможный метод записи этих чисел состоит в использовании формата «градусы/десятичные минуты»:

```
176° 14.066'
```

Наконец, есть формат «десятичные градусы»:

```
176.234436°
```

Десятичные градусы сегодня получили бóльшую распространенность, главным образом потому, что это просто числа с плавающей точкой, которые можно вводить прямиком в программы, правда, прежде чем их можно будет использовать, вам вполне может понадобиться сначала конвертировать значения долготы и широты из других форматов.

Еще одна возможная проблема, связанная со значениями долготы и широты, заключается в том, что «квадрант» (восток, запад, север или юг) может иногда задаваться как разностная величина, а не как положительная или отрицательная величина. Вот пример:

```
176.234436° E
```

---

[1] Один радиан – это примерно 57°. В одном полном обороте ровно 2π радиан. – *Прим. перев.*

К счастью, все эти преобразования относительно прямолинейны. Однако важно знать, в каких единицах и каком формате ваши данные находятся, – ваше программное обеспечение, может, и не разрушит космический корабль, но, если вы не будете осмотрительны, оно вполне может привести к некоторым очень странным и необъяснимым результатам.

Картографические проекции

Процесс создания двумерной карты из трехмерной фигуры Земли именуется картографической проекцией. **Картографическая проекция** – это математическое преобразование, которое «развертывает» трехмерную фигуру Земли и помещает ее на двумерную плоскость.

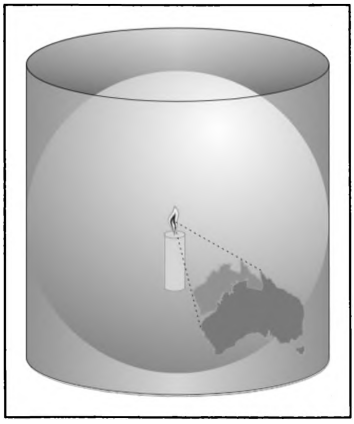
Существуют сотни различных картографических проекций, но ни одна из них не совершенна. И действительно, математически невозможно представить трехмерную поверхность Земли на двумерной плоскости, не вводя своего рода искажения; фокус состоит в том, чтобы выбрать такую проекцию, при которой искажение никак не сказывается на вашей работе. Например, некоторые проекции точно обозначают конкретные участки земной поверхности, добавляя главное искажение в другие ее участки; такие проекции годятся для картирования строго определенного участка Земли и никакого другого. Еще одни проекции искажают фигуру страны, оставляя ее площадь без изменений, в то время как другие проекции делают противоположное.

По характеру искажений различают три основные группы картографических проекций: цилиндрические, конические и плоскостные. Коротко рассмотрим каждую из них.

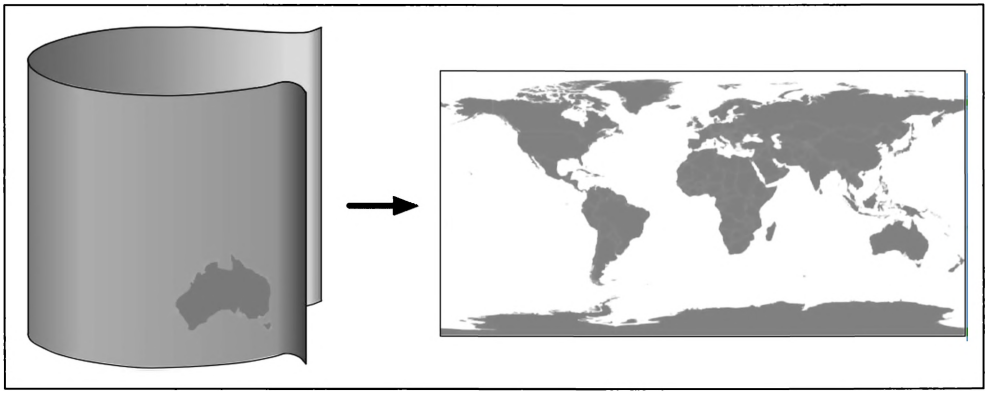
Цилиндрические проекции

Есть простой способ разобраться в цилиндрических проекциях – надо представить, что Земля выглядит как сферический китайский фонарь со свечой в середине:

Если поместить эту землю-фонарь в бумажный цилиндр, то свеча «спроецирует», то есть перенесет, земную поверхность на внутреннюю часть цилиндра:

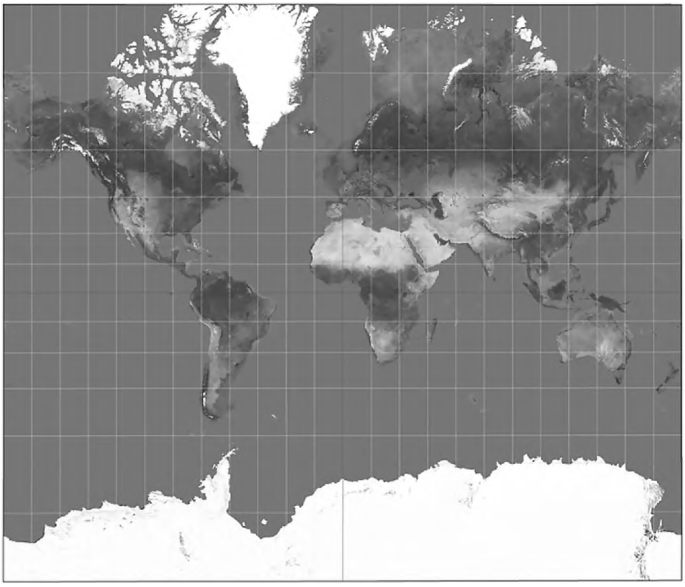


И тогда этот цилиндр можно «развернуть», чтобы получить двумерное изображение Земли:



Разумеется, это упрощение – на самом деле в картографических проекциях источники света для отображения земной поверхности на плоскость вовсе не используются. Чтобы достичь того же самого эффекта, вместо этого используются сложные математические преобразования.

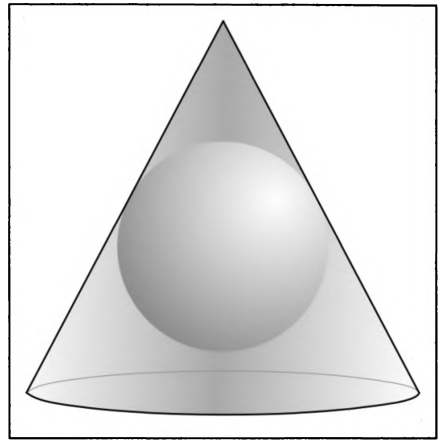
Вот некоторые главные типы цилиндрических проекций: **проекция Меркатора**, **равновеликая цилиндрическая проекция** и **универсальная поперечная проекция Меркатора (UTM)**[1]. Следующая ниже карта взята из Википедии и представляет собой образец проекции Меркатора:



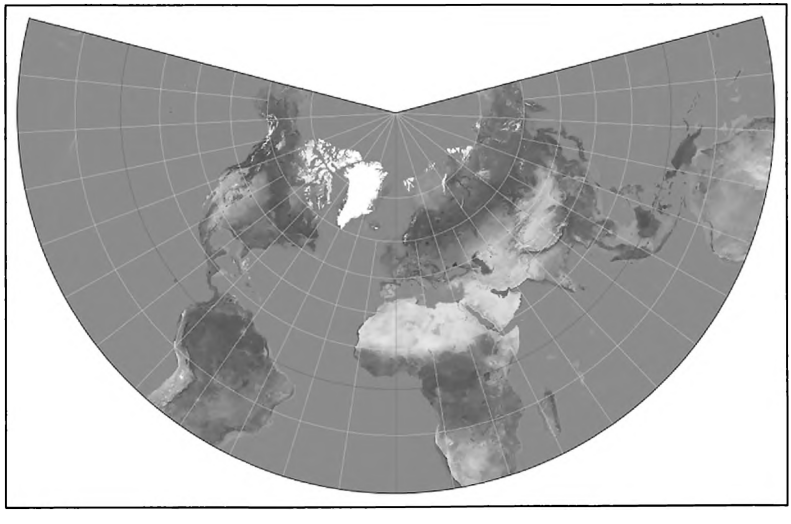
[1] Соответствующие термины на английском языке: Mercator projection, equal-area cylindrical projection и universal transverse Mercator projection. – *Прим. перев.*

Конические проекции

Коническую проекцию получают путем отображения земной поверхности на конус:



Затем конус «разворачивают», чтобы получить окончательную карту. Вот несколько распространенных типов конических проекций: **равновеликая проекция Алберса**, **равноугольная (конформная) коническая проекция Ламберта** и **равноудаленная проекция**[1]. Ниже приведен пример конформной конической проекции Ламберта, снова взятый из Википедии:

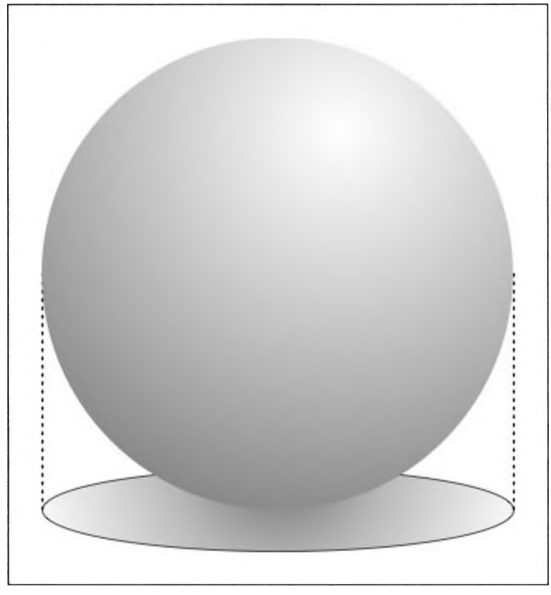


[1] Соответствующие термины на английском языке: Albers equal-area projection, Lambert conformal conic projection и equidistant projection. – *Прим. перев.*

Полярно выровненные конические проекции особенно годятся для отображения широких и не очень высоких участков Земли, как, например, карта России.

Плоскостные проекции

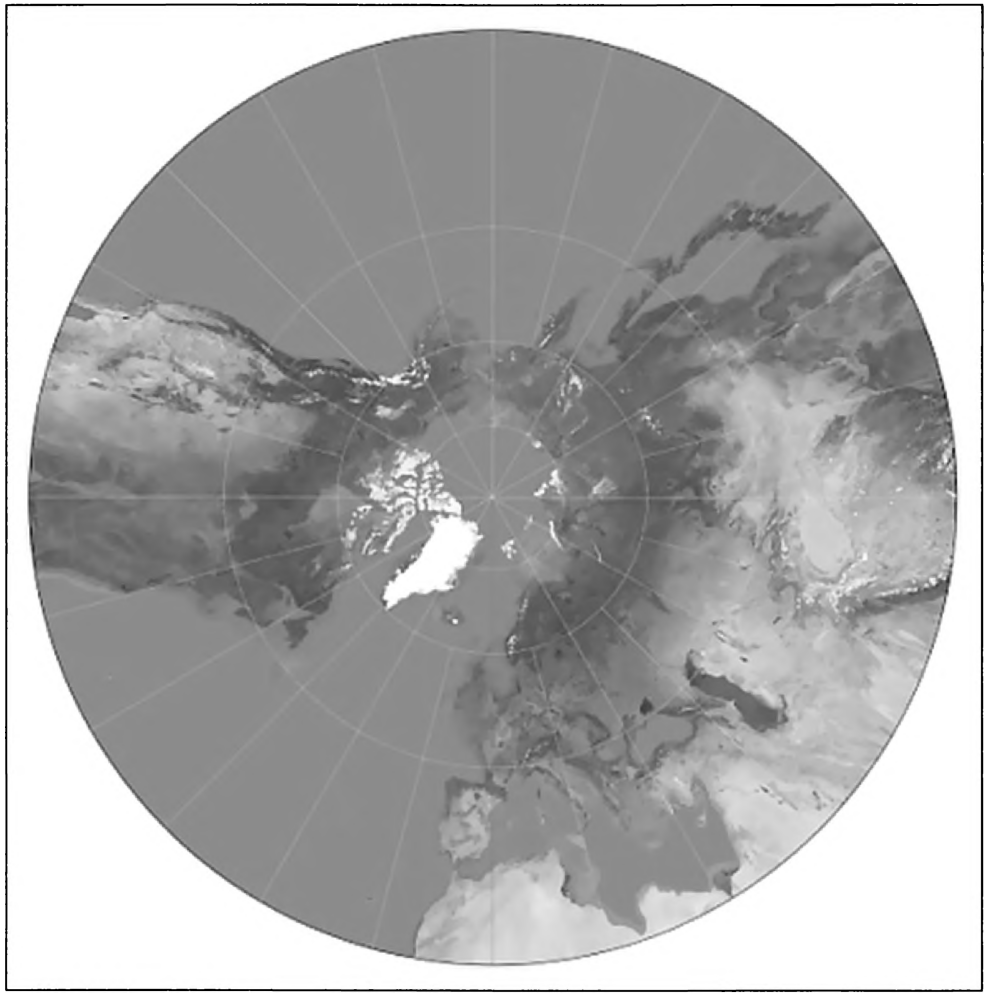
Плоскостная, или азимутальная, проекция подразумевает отображение поверхности Земли непосредственно на плоскость:



Плоскостные проекции центрируются вокруг единственной точки и обычно поверхности всей Земли не показывают. Однако же они подчеркивают сферическую природу Земли. По многим аспектам плоскостные проекции изображают Землю такой, какой ее можно было бы наблюдать из космоса.

Вот несколько основных типов плоскостных проекций: **гномоническая проекция**, **равновеликая азимутальная проекция Ламберта** и **ортогональная проекция**[1]. Следующий пример, взятый из Википедии, демонстрирует гномоническую проекцию, базирующуюся вокруг Северного полюса:

[1] Соответствующие термины на английском языке: gnomonic projection, Lambert equal-area azimuthal projection и orthographic projection. – *Прим. перев.*



Природа картографических проекций

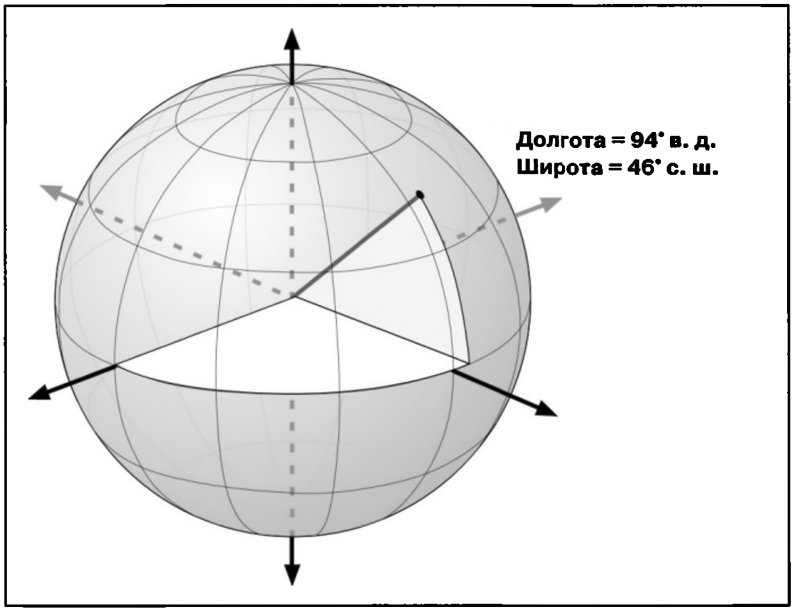
Как отмечалось ранее, совершенной проекции в природе не существует – каждая проекция некоторым образом искажает земную поверхность. И действительно, математик Карл Гаусс доказал, что математически невозможно спроецировать трехмерную геометрическую фигуру, такую как сфера, на плоскость, не введя неких искажений. Именно по этой причине существует столько разных типов картографических проекций: некоторые проекции больше подходят для какой-то конкретной цели, но никакая проекция не может охватить все.

Во время создания геопространственных данных или работы с ними всегда важно знать, какая картографическая проекция применялась при создании этих данных. Не зная проекции, вы не сможете выполнять ни визуализации данных, ни точных расчетов.

Системы координат

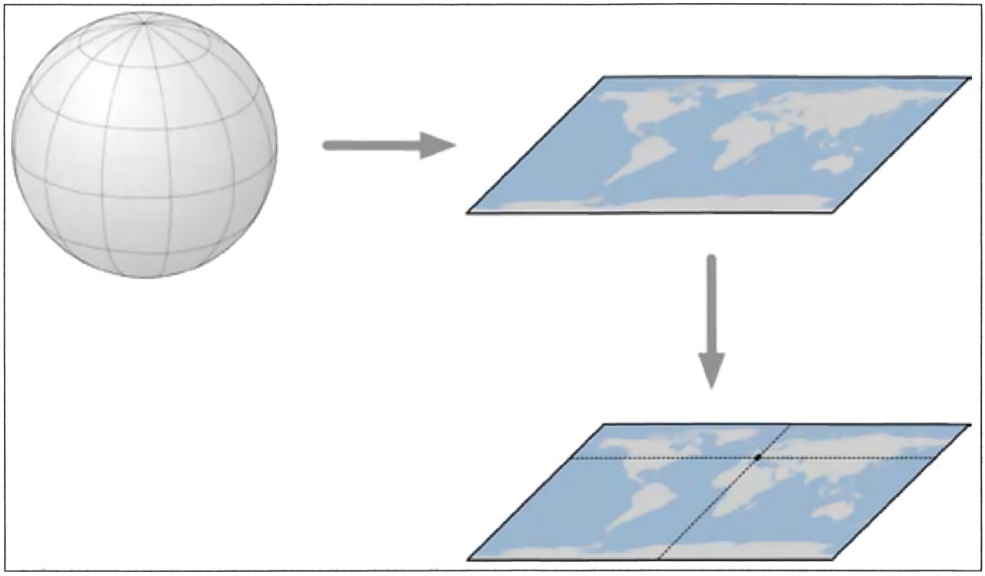
Понятие **системы координат** тесно связано с картографическими проекциями. Есть два типа систем координат, в которых необходимо хорошо разбираться: **картографические** (спроецированные) и **географические** (неспроецированные).

Примером географической системы координат являются значения широты и долготы. Эти географические координаты прямо указывают на точку, расположенную на поверхности Земли:



Географические координаты полезны тем, что они могут точно представлять конкретную точку на поверхности Земли. Вместе с тем они сильно усложняют расчеты расстояния и выполнение других геопространственных вычислений.

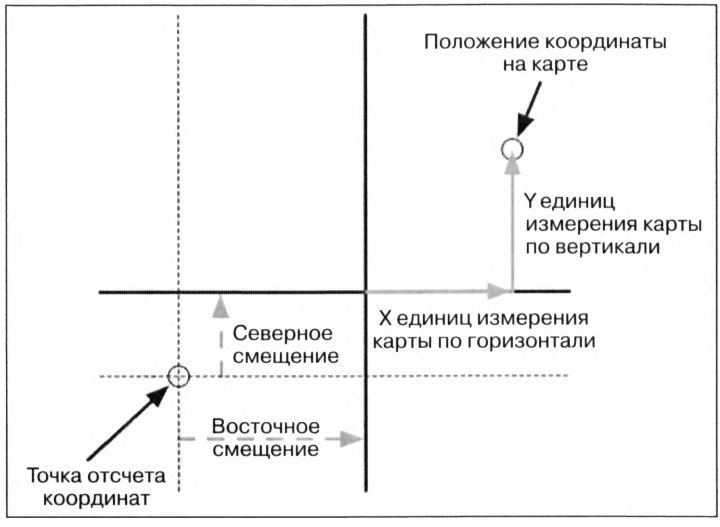
В отличие от них, в картографической системе координат прямоугольные координаты указывают на точку, расположенную на двумерной карте, которая *изображает* поверхность Земли:



В картографической (прямоугольной) системе координат, как понятно из названия, сначала используется картографическая проекция, которая преобразует Землю в двумерную (картезианскую) координатную плоскость, после чего на эту плоскость помещаются точки. Чтобы работать с картографической системой координат, нужно знать, какая проекция использовалась для создания лежащей в основе карты.

В картографических и географических системах координат, кроме того, предполагается наличие набора **начальных (опорных) точек** (reference points), которые позволяют определять, где конкретная точка находится. Например, в географической системе координат (широты и долготы) нулевое значение долготы представлено нулевым (гринвичским) меридианом, то есть линией, проходящей с севера на юг через Гринвичскую обсерваторию в Англии. Аналогичным образом нулевое значение широты выражено нулевой параллелью, то есть линией, проходящей вокруг экватора.

В отличие от них, для картографических систем координат обычно задают **начало отсчета координат и единицы измерения карты**. В некоторых системах координат также используются значения **северного смещения** (false northing) и **восточного смещения** (false easting), чтобы скорректировать положение начала отсчета координат, как показано ниже:



Покажем на конкретном примере. Система координат **универсальной поперечно-цилиндрической проекции Меркатора (UTM)** делит мир по меридианам на 60 «зон», в каждой из которых с целью минимизации ошибки проекции используется другая картографическая проекция. В пределах конкретной зоны координаты измеряются как количество метров от ее начала отсчета координат, то есть пересечения экватора с центральным меридианом для этой зоны. И затем значения северного смещения и восточного смещения в метрах прибавляются к расстоянию от точки отсчета, позволяя избежать ситуаций, когда приходится иметь дело с отрицательными числами.

Как вы понимаете, работа с подобными картографическими (спроецированными) системами координат может стать довольно запутанной. Однако большое преимущество прямоугольных координат состоит в том, что, используя эти координаты, можно легко выполнять геопространственные расчеты. Например, чтобы рассчитать расстояние между двумя точками, которые используют одну и ту же систему координат UTM, надо просто вычислить длину отрезка между ними. Ее длина – это расстояние между двумя точками в метрах. Этот процесс до смешного прост, по сравнению с работой, которую требуется проделать, чтобы рассчитать расстояние при помощи географических (неспроецированных) координат.

Конечно же, мы исходим из предположения, что обе точки находятся в одинаковой системе координат. Поскольку картографические системы координат вообще-то имеют точность только на относительно небольшом участке, то можно попасть в беду, если обе точки не находятся в одинаковой системе координат (например, если они находятся в двух разных зонах UTM). Как раз тут и проявляется большое преимущество географических систем координат: они охватывают всю Землю.

Геодезические датумы

В общих чертах **геодезический датум**[1] – это математическая модель Земли, которая используется для описания местоположений на ее поверхности. Датум состоит из набора опорных точек, или базовых геодезических параметров, часто в комбинации с моделью фигуры Земли. Опорные точки используются для описания местнахождения других точек на поверхности Земли, в то время как модель фигуры Земли используется при отображении поверхности Земли на двумерную плоскость карты. Таким образом, датумы используются как в картографических проекциях, так и в системах координат.

Хотя во всем мире количество находящихся в употреблении датумов насчитывается сотнями, все же большинство из них имеет отношение исключительно к локальным территориям, и только три считаются основными. Это так называемые **опорные датумы** (reference datums), которые охватывают более обширные территории и с которыми вы, вероятно, встретитесь, когда будете работать с геоданными:

- **NAD 27**: североамериканский датум 1927 года (North American Datum). Он состоит из определения фигуры Земли (используя модель под названием Сфероид Кларка 1866) и набора базовых параметров, центрированных вокруг геодезического центра Ранчо Мида (Meades) в шт. Канзас. NAD 27 можно рассматривать как локальный датум с зоной покрытия, включающей Северную Америку;
- **NAD 83**: североамериканский датум 1983 года. В нем использована более сложная модель фигуры Земли (геодезическая опорная система 1980 года, GRS 80). NAD 83 можно считать локальным датумом с зоной покрытия, включающей США, Канаду, Мексику и Центральную Америку;
- **WGS 84**: мировая геодезическая система 1984 (World Geodetic System). Это глобальны датум, чья зона покрытия – вся поверхность Земли. В нем использована еще одна модель фигуры Земли (гравитационная модель Земли 1996, EGM 96) и базовые параметры на основе Международного опорного меридиана IERS. Датум WGS 84 очень популярен. При работе с геоданными, охватывающими территорию США, WGS 84 практически совпадает с NAD 83. Правда, в отличие от него, WGS 84 используется спутниками **Системы глобального позиционирования** GPS, и поэтому все данные, полученные от модулей GPS, будут использовать этот датум.

Отечественной альтернативой WGS 84 является датум ПЗ-90 (Параметры Земли 1990 года) – это система базовых геодезических параметров, включающая фундаментальные геодезические постоянные, параметры общеземного эл-

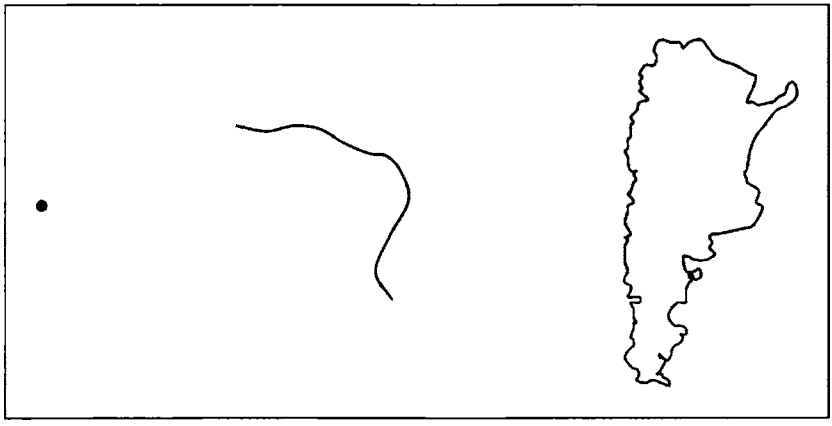
[1] В русскоязычной литературе вместо термина «геодезический датум» часто используются термины «исходная геодезическая дата» или «геодезическая отсчетная основа». – *Прим. перев.*

липсоида, параметры гравитационного поля Земли, геоцентрическую систему координат и параметры ее связи с другими системами координат. Используется в целях геодезического обеспечения орбитальных полетов и решения навигационных задач (в частности, для обеспечения работы глобальной навигационной спутниковой системы ГЛОНАСС).

На сегодняшний день самым распространенным датумом является глобальный датум WGS 84, вместе с тем во многих геоданных предполагается использование других датумов. Имея дело со значением координаты, всегда важно знать, какой датум использовался во время расчета этой координаты. К примеру, конкретная точка, выраженная в датуме NAD 27, может быть на расстоянии в сотни футов от той же самой координаты, выраженной в датуме WGS 84. Таким образом, жизненно важно знать, какой датум используется для конкретного набора геоданных, и в случае необходимости конвертировать в другой датум.

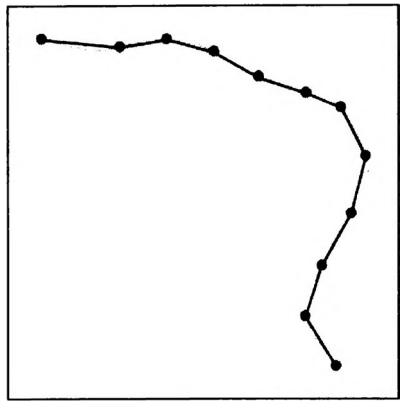
Географические фигуры

Геоданные нередко содержат *географические фигуры* (shapes), точнее характерный профиль или вид геопространственных объектов, представленных на карте. Географические фигуры представлены точками, кривыми и контурами:



Разумеется, **точка** – это просто координата, описываемая как минимум двумя числами, в пределах картографической или географической системы координат.

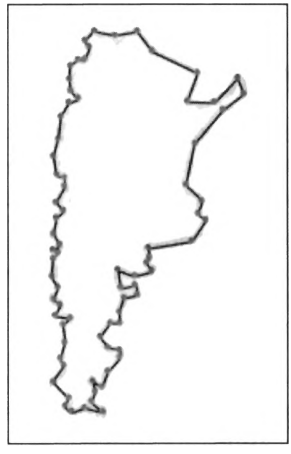
Кривая обычно описывается списком значений, именуемым **ломаной линией**, или полилиней:



Ломаная представляет кривую в виде связанного множества отрезков. Будучи намеренно упрощенным представлением кривой, ломаная линия дает возможность аппроксимировать криволинейную траекторию без необходимости иметь дело со сложными математическими расчетами, которые требуются для начертания кривых и управления ими. Ломаные часто используются в геоданных для описания дорог, рек, изолиний и т. д.

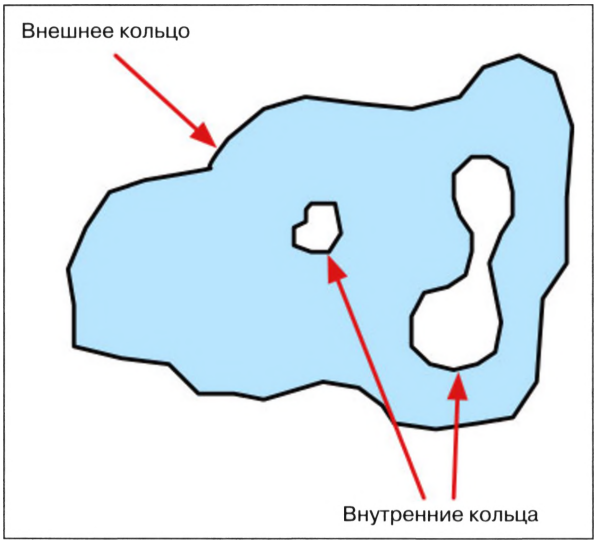
В англоязычной литературе для термина «ломаная» используется слово PolyLine или его синоним LineString. Замкнутая ломаная (то есть когда последний отрезок заканчивается в точке, где начинается первый отрезок) часто упоминается как линейное кольцо[1] (linear ring).

В геоданных контур часто представлен **многоугольником**, или полигоном:



[1] Линейное кольцо (linear ring) – это замкнутая и одновременно простая ломаная линия. Другими словами, первая и последняя координаты в кольце должны быть одинаковыми, а внутри кольца не должно быть самопересечений. – *Прим. перев.*

Многоугольники в геоданных обычно используются для описания контуров стран, озер, городов и т. д. Многоугольник имеет **внешнее кольцо**, задаваемое замкнутой ломаной, и может дополнительно иметь одно или несколько вложенных **внутренних колец**, каждое из которых тоже задано замкнутой ломаной линией. Внешнее кольцо представляет контур многоугольника, в то время как внутренние кольца (если таковые имеются) представляют «дыры» внутри многоугольника:



Эти дыры часто используются для изображения внутренних элементов, таких как острова на озере.

Форматы данных ГИС

Формат данных ГИС определяет, каким образом геоданные хранятся в файле (либо нескольких файлах) на диске, и описывает логическую структуру, используемую для хранения геоданных внутри файла (файлов).

Хотя мы говорим о хранении информации на диске, вместе с тем форматы данных используются и для передачи геоданных между вычислительными системами. Так, веб-служба предоставляет данные географической карты по запросу, передавая их в заданном формате.

Формат данных ГИС, как правило, поддерживает следующие данные:

- геопространственные данные с описанием географических **объектов**, или геообъектов[1], что то же самое;

[1] Геообъект – элемент данных, представляющий собой модель географического объекта реального мира, которая задается геометрией, атрибутами и координатами. Термины *пространственный объект* и *геообъект* используются как синонимы. – *Прим. перев.*

- дополнительные **метаданные**, которые дают описание самих геопространственных данных, в том числе используемые датум и проекцию, систему координат и единицы измерения данных, дату последнего обновления файла данных и т. д.;
- **атрибуты**, которые предоставляют дополнительную информацию об описываемых географических объектах. Например, геообъект, обозначающий город, может иметь такие атрибуты, как название, население, средняя температура и т. д.;
- **визуально отображаемая информация**, в том числе цвет или стиль линии, используемые при изображении объекта.

В ГИС используются два главных типа данных: **данные в растровом формате** и **данные в векторном формате**. Растровые форматы обычно используются для хранения побитово адресуемых графических изображений, таких как отсканированные бумажные карты или аэрофото- и космические снимки. В свою очередь, векторные форматы представляют пространственные данные при помощи точек, линий и многоугольников. Использование векторных форматов в ГИС-приложениях более распространено, поскольку такие данные компактнее и ими легко управлять.

Вот несколько наиболее распространенных растровых форматов:

- **DRG** (Digital Raster Graphic) – формат цифровой растровой графики, используется для хранения цифровых сканов бумажных карт;
- **DEM** (Digital Elevation Model) – формат цифровой модели местности, используется для записи данных высот;
- **BIL** (Band Interleaved by Line) с построчным чередованием, **BIP** (Band Interleaved by Pixel) с попиксельным чередованием, **BSQ** (Band Sequential) с последовательным чередованием каналов – эти форматы данных обычно используются системами дистанционного зондирования[1].

Приведем ряд самых распространенных векторных форматов:

- **файл фигур**, или shape-файл[2] – открытая спецификация, разработанная американской компанией «Институт исследования систем окружающей среды», англ. название Environmental Systems Research Institute (ESRI), для хранения и обмена данными ГИС. Файл фигур на деле состоит из на-

---

[1] Форматы BSQ, BIL и BIP предлагают способы хранения 24-разрядных изображений, у которых всегда имеются 3 канала, обозначающих красную, зеленую и синюю плоскости изображения. Но «тройка» в размерности изображения может стоять в различных положениях (*xy3*, *3xy*, *x3y*), где x – это размерность столбца (число столбцов в изображении), y – размерность строки, а размерность 3 равна числу каналов в изображении. BIL соответствует x3y, BIP соответствует 3xy, и BSQ соответствует xy3. – *Прим. перев.*

[2] Векторный формат хранения данных о географическом положении, геометрии и атрибутах геообъектов; содержит один класс геообъектов. В отечественной литературе по ГИС файлы с расширением .shp называются по-разному: shape-файлы, SHP-файлы, файлы карт, файлы форм. В данном переводе за основу принят термин *файл фигур*, используемый для таких файлов в AutoCAD. – *Прим. перев.*

бора файлов с одинаковым базовым именем, например `hawaii.shp`, `hawaii.shx`, `hawaii.dbf` и т. д.;

- **Simple Features** (переводится как *простые геообъекты*) – стандарт OpenGIS для хранения географических данных (точек, линий и многоугольников) вместе со связанными с ними атрибутами;
- **TIGER/Line** – текстовый формат, ранее использовавшийся Бюро переписи населения США, англ. название US Census Bureau (USCB), служит для описания географических объектов, таких как дороги, здания, реки и береговые линии. Свежие данные поступают в формате файла фигур, и поэтому формат TIGER/Line в основном используется для более ранних наборов данных этой организации;
- **Coverage** (переводится как *пространственное покрытие*) – собственный формат данных, используемый в системе ARC/INFO американской компании ESRI для хранения географических объектов в виде точек, дуг и многоугольников со связанными с ними таблицами атрибутов геообъектов.

В дополнение к этим «важным» форматам данных еще существуют так называемые **микроформаты**, которые часто используются для описания отдельных сегментов геоданных. По большей части они используются для описания фигуры геообъекта внутри рабочей программы или передачи его описания из одной программы в другую и для постоянного хранения данных обычно не применяются.

Работая с геоданными, вы, скорее всего, столкнетесь со следующими микроформатами:

- **WKT** (Well-known Text), или стандартное текстовое представление, – простой текстовый формат для представления одного географического объекта, такого как многоугольник или ломаная;
- **WKB** (Well-known Binary), или стандартное двоичное представление, – альтернатива WKT; вместо текста использует двоичные данные; служит для представления одного географического объекта[1];
- **GeoJSON** – открытый формат для кодирования географических структур данных, который основан на формате для обмена данными JSON;
- **GML** (Geography Markup Language), или язык географической разметки, – открытый стандарт на основе XML для обмена данными ГИС.

Работая с геоданными, всегда нужно знать, в каком формате данные находятся, с тем чтобы из файла(ов) можно было извлечь информацию, в которой вы нуждаетесь, и в случае необходимости перевести данные из одного формата в другой.

Работа с данными ГИС вручную

Коротко ознакомимся с процессом работы с данными ГИС в ручном режиме. Перед началом работы необходимо выполнить две вещи:

---

[1] Форматы WKT и WKB используются для представления геометрий и описания систем координат геообъекта. – *Прим. перев.*

- получить некие данные ГИС;
- установить библиотеку GDAL и привязки Python, с тем чтобы можно было читать необходимые файлы данных.

Получение данных

Воспользуемся веб-сайтом Бюро переписи населения США, чтобы скачать набор векторных карт штатов США. Главный сайт Бюро переписи населения США, где можно получить данные ГИС, находится на странице http://www.census.gov/geo/www/tiger.

Впрочем, можно упростить задачу и пойти обходным путем, скачав требующийся файл по прямой ссылке: http://www2.census.gov/geo/tiger/TIGER2014/STATE/tl_2014_us_state.zip.

В ряде случаев прямой доступ из домена ru к американским правительственным сайтам затруднен. В таких случаях следует пользоваться анонимайзером, то есть специальным веб-сайтом для анонимного доступа, как, например, Хамелеон (http://cameleo.ru/). В данном случае, чтобы скачать нужный файл с сайта Бюро переписи США, в текстовое поле без пробелов скопируйте следующую далее часть URL-адреса: http://www2.census.gov/geo/tiger/TIGER2014/STATE. В результате вы попадете на страницу скачивания нужного вам файла. То же самое можно сделать, пройдя вниз по иерархии каталогов, если наберете адрес http://www2.census.gov/.

Файл tl_2014_us_state.zip – это архив в формате ZIP. После распаковки архива у вас должен появиться каталог, в котором содержатся следующие файлы:

- tl_2014_us_state.dbf;
- tl_2014_us_state.prj;
- tl_2014_us_state.shp;
- tl_2014_us_state.shx.

Эти файлы составляют файл фигур, содержащий контуры всех американских штатов. Поместите эти файлы все вместе в один подходящий каталог.

Кроме них, еще будет несколько файлов с расширением .xml. Они содержат дополнительную информацию о загруженных данных, но при этом не являются частью самого файла фигур.

Инсталляция библиотеки GDAL

Затем нужно скачать и установить динамическую библиотеку[1] GDAL и файлы привязок Python. Главный веб-сайт библиотеки GDAL находится по адресу http://gdal.org.

---

[1] Динамическая (разделяемая) библиотека – файл с расширением .so, .dll либо .dylib, в котором хранится скомпилированный программный код, написанный на C/C++/Fortran и содержащий критические в плане скорости исполнения процедуры и специализированные функции, которые позволяют расширить производительность и функциональность рабочей программы. Загружается в память по запросу уже работающего процесса, то есть динамически. – *Прим. перев.*

Способ инсталляции библиотеки зависит от операционной системы, в которой вы работаете.

Инсталляция библиотеки GDAL в Linux

Для установки библиотеки GDAL на компьютер под Linux обычно используют диспетчер пакетов, который устанавливает ее сам. Например, в Fedora 24[1] используется следующая команда:

```
dnf install gdal gdal-devel
```

Отметим, что нужно установить пакет gdal-devel, а также саму библиотеку gdal. Пакет gdal-devel содержит файлы, необходимые для компиляции привязок Python для библиотеки GDAL.

После установки библиотеки GDAL нужно установить и собрать привязки Python. Программный код привязок Python находится на странице https://pypi.python.org/pypi/gdal. Как правило, используется нижеследующая команда:

```
sudo easy_install GDAL
```

Разумеется, прежде чем устанавливать привязки Python для библиотеки GDAL, среда Python должна быть уже установлена.

Все используемые в книге библиотеки можно найти в репозиториях пакетов RPM для операционных систем Linux, таких как репозиторий пакетов RPM rpmfind (http://rpmfind.net/linux/RPM/ и на его зеркале https://www.rpmfind.net/), на веб-сайте Koji, посвященном сборке пакетов для проекта Fedora (http://koji.fedoraproject.org/koji/), и на странице проекта, посвященного пакетированию ГИС-ориентированного ПО (https://fedoraproject.org/wiki/GIS). Например, библиотеку GDAL можно скачать по ссылке https://www.rpmfind.net/linux/rpm2html/search.php?query=gdal-python3 и затем установить стандартным образом (fc24 соответствует Fedora 24):

```
dnf install /путь/к/файлу/gdal-python3-2.0.2-6.fc24.x86_64.rpm
```

Так или иначе, в конце следует проверить инсталляцию. Откройте оболочку Python 3 (IDLE3) или инструментальную среду Spyder и в терминале наберите `import gdal`. Отсутствие сообщения об ошибке будет свидетельствовать об успешности установки привязок Python. По поводу инсталляции инструментальной среды Spyder см. «*Комментарий переводчика*».

Инсталляция библиотеки GDAL в Mac OS X

Для инсталляции библиотеки GDAL в Mac OS X необходимо установить саму библиотеку GDAL и затем скомпилировать привязки Python, чтобы использовать установленную версию GDAL. Разберем по очереди этапы процесса инсталляции:

---

[1] Fedora – дистрибутив операционной системы GNU/Linux, который служит для тестирования новых технологий, в дальнейшем включаемых в продукты Red Hat и других производителей. – *По материалам Википедии.*

1. Чтобы скачать библиотеку GDAL, перейдите по ссылке на http://www.kyngchaos.com/software/frameworks. Необходимо установить *GDAL Complete*, то есть полнофункциональный пакет. Этот пакет содержит предварительно скомпилированную версию динамической библиотеки GDAL, которую следует установить на компьютер.
2. После инсталляции библиотеки GDAL затем нужно скомпилировать привязки Python. Для этого сначала необходимо установить среду разработки Apple XCode. Среду XCode можно скачать бесплатно из магазина Mac App Store.

Отметим, что если ваш компьютер работает под Mac OS X версии ниже 10.9 (Yosemite), то вам придется установить инструменты командной строки для XCode отдельно. Для этого запустите среду XCode и выберите из меню Xcode команду Настройки. На вкладке Загрузки будет находиться опция установки инструментов командной строки; активируйте эту опцию и подождите, пока будут установлены необходимые инструменты.

3. После установки среды XCode можно скачать и скомпилировать привязки Python для библиотеки GDAL. Самый простой способ – использовать диспетчер пакетов Python **pip**. Если у вас его нет, то его можно установить, следуя инструкциям на странице https://pip.pypa.io/en/latest/installing.html.
4. Прежде чем скомпилировать привязки Python, необходимо задать несколько переменных окружения, с тем чтобы Python мог найти установленную версию GDAL. Для этого в окне терминала введите следующие команды:

```
export CPLUS_INCLUDE_PATH=/Library/Frameworks/GDAL.framework/Headers
export C_INCLUDE_PATH=/Library/Frameworks/GDAL.framework/Headers
export LIBRARY_PATH=/Library/Frameworks/GDAL.framework/Versions/1.11/unix/lib
```

   Удостоверьтесь, что версия GDAL в определении переменной library_path соответствует версии, которую вы установили.
5. Теперь можно установить привязки Python, набрав нижеследующее:

```
pip install gdal==1.11
```

   Еще раз убедитесь, что номер версии соответствует установленной версии GDAL.

Если все выполнено правильно, то привязки Python для библиотеки GDAL должны скомпилироваться успешно. При компиляции будет выведено несколько предупреждений компилятора и, будем надеяться, никаких ошибок.

Инсталляция библиотеки GDAL в Windows

К сожалению, в настоящее время предварительно скомпилированный установщик для GDAL, который использует Python 3, отсутствует. Можно попытаться собрать его самостоятельно либо придется использовать Python 2. Двоичный установщик для GDAL, который поддерживает Python 2, можно скачать с сайта http://www.gisinternals.com.

Так как вы не сможете использовать Python 3, вам придется приспособить ваш программный код таким образом, чтобы он соответствовал синтаксису Python 2.

Что касается примеров программного кода для этой главы, то единственное отличие состоит в том, что после ключевого слова print круглые скобки не используются.

На момент перевода книги (август-сентябрь 2016 г.) появилась поддержка Python 3 в Windows. Чтобы установить библиотеку GDAL версии 2.0.3 для Python 3, нужно скачать wheel-файл GDAL-2.0.3-cp35-cp35m-win_amd64.whl со страницы Кристофа Голька из Лаборатории динамики флуоресценции Калифорнийского университета в г. Ирвайн, на которой, кстати, приведен обширный перечень неофициальных бинарников для питоновских пакетов расширения (http://www.lfd.uci.edu/~gohlke/pythonlibs/#gdal). Затем скопировать скачанный файл в папку C:\Users\labor\Anaconda3\Scripts, запустить инструментальную среду программирования Spyder и набрать в окне терминала команду

```
pip install GDAL-2.0.3-cp35-cp35m-win_amd64.whl
```

После установки библиотеки ее работоспособность можно проверить, набрав команду import gdal в окне консоли Python возле приглашения на ввод команд >>>, и если никаких сообщений не последовало, то установка библиотеки прошла успешно. Работа программ была протестирована в Windows 10. По поводу инсталляции инструментальной среды Spyder см. «*Комментарий переводчика*».

Тестирование библиотеки GDAL

После установки библиотеки GDAL можно проверить ее работоспособность, набрав в командной строке Python команду import osgeo; если приглашение командной строки Python появилось вновь без сообщения об ошибке, то, значит, библиотека GDAL была успешно установлена, и все готово к работе:

```
>>> import osgeo
>>>
```

Исследование скачанного файла фигур

Теперь воспользуемся библиотекой GDAL, чтобы взглянуть на содержимое файла фигур, который вы скачали ранее. Следующий ниже пример можно набрать непосредственно в командной строке либо сохранить в виде сценария Python, чтобы выполнять его в любое время (назовем этот сценарий analyze1.py):

```
import osgeo.ogr

shapefile = osgeo.ogr.Open("tl_2014_us_state.shp")
numLayers = shapefile.GetLayerCount()
print("Файл фигур содержит {} слой(ев)".format(numLayers))
print()

for layerNum in range(numLayers):
  layer = shapefile.GetLayer(layerNum)
  spatialRef = layer.GetSpatialRef().ExportToProj4()
  numFeatures = layer.GetFeatureCount()
  print("Слой {} имеет пространственную привязку {}".format(
        layerNum, spatialRef))
```

```
print("Слой {} содержит {} геообъектов:".format(
      layerNum, numFeatures))
print()

for featureNum in range(numFeatures):
  feature = layer.GetFeature(featureNum)
  featureName = feature.GetField("NAME")

  print("Геообъект {} под названием {}".
        format(featureNum, featureName))
```

Этот пример предполагает, что вы поместили сценарий в тот же самый каталог, что и файл `tl_2 014_us_state.shp`. Если вы поместили его в другой каталог, то внесите изменения в команде `osgeo.ogr.Open()`, чтобы указать путь к вашему файлу фигур. Если сценарий выполняется в Windows, то напоминаем об использовании двойной обратной косой черты (\\) в качестве символа-разделителя папок.

Выполнение этого сценария выведет краткое резюме о том, каким образом данные файла фигур структурированы:

```
Файл фигур содержит 1 слой(ев)
Слой 0 имеет пространственную привязку +proj=longlat +datum=NAD83 +no_defs
Слой 0 содержит 56 геообъектов:
Геообъект 0 называется West Virginia
Геообъект 1 называется Florida
Геообъект 2 называется Illinois
Геообъект 3 называется Minnesota
...
Геообъект 53 называется District of Columbia
Геообъект 54 называется Iowa
Геообъект 55 называется Arizona
```

В резюме сообщается, что скачанные данные состоят из одного слоя с 56 отдельными геообъектами, которые соответствуют различным штатам и протекторатам США. В нем также говорится об используемой в этом слое так называемой *пространственной привязке* (spatial reference), которая сообщает, что координаты представлены значениями широты и долготы на основе геодезического датума NAD83.

Как видите, применение библиотеки GDAL для извлечения данных из файлов фигур достаточно прямолинейное. Продолжим на другом примере. На этот раз обратимся к описанию объекта номер 12, шт. Нью-Мексико. Назовем эту программу `analyze2.py`:

```
import osgeo.ogr

shapefile = osgeo.ogr.Open("tl_2014_us_state.shp")
layer = shapefile.GetLayer(0)
feature = layer.GetFeature(12)

print("Геообъект № 12 имеет следующие атрибуты:")
print()
```

```
attributes = feature.items()

for key,value in attributes.items():
    print(" {} = {}".format(key, value))

geometry = feature.GetGeometryRef()
geometryName = geometry.GetGeometryName()

print()
print("Геометрия заданного геообъекта представляет собой {}".format(geometryName))
```

В результате выполнения этого сценария будет получен следующий результат:

```
Геообъект № 12 имеет следующие ниже атрибуты:

STUSPS = NM
GEOID = 35
REGION = 4
ALAND = 314161410324.0
MTFCC = G4000
FUNCSTAT = A
NAME = New Mexico
STATEFP = 35
LSAD = 00
STATENS = 00897535
AWATER = 755712514.0
DIVISION = 8
INTPTLON = -106.1316181
INTPTLAT = +34.4346843

Геометрия заданного геообъекта представляет собой POLYGON
```

Описание значений различных атрибутов приведено на веб-сайте Бюро переписи населения США, однако прямо сейчас нас интересует **геометрия** пространственного объекта[1]. Геометрия – это объект со сложной и нередко рекурсивной структурой, отражающей характер организации геопространственных данных. Пока что мы выяснили, что шт. Нью-Мексико имеет форму многоугольника. Теперь приглядимся к этому многоугольнику поближе при помощи программы, которую мы назовем `analyze3.py`:

```
import osgeo.ogr

def analyzeGeometry(geometry, indent=0):
    s = []
    s.append(" " * indent)
    s.append(geometry.GetGeometryName())
    if geometry.GetPointCount() > 0:
        s.append(" с {} точками данных".format(geometry.GetPointCount()))
```

[1] Пространственный объект (feature), или геообъект, – это цифровое представление объекта реального мира на карте, которое содержит пространственную привязку (описание геометрии) и набор атрибутов (текстовых и числовых характеристик). – *Прим. перев.*

```
    if geometry.GetGeometryCount() > 0:
        s.append(" содержит:")

    print("".join(s))

    for i in range(geometry.GetGeometryCount()):
        analyzeGeometry(geometry.GetGeometryRef(i), indent+1)

shapefile = osgeo.ogr.Open("tl_2014_us_state.shp")
layer = shapefile.GetLayer(0)
feature = layer.GetFeature(12)
geometry = feature.GetGeometryRef()

analyzeGeometry(geometry)
```

Рекурсивная функция `analyzeGeometry()` раскрывает структуру его геометрии:

```
POLYGON содержит:
  LINEARRING с 7554 точками данных
```

Как видите, приведенная геометрия имеет единственное линейное кольцо, которое задает контур шт. Нью-Мексико. Если бы мы выполнили эту же самую программу для шт. Калифорния (объект 13 в файле фигур), то результат был бы несколько более сложным:

```
MULTIPOLYGON содержит:
  POLYGON содержит:
    LINEARRING с 121 точкой данных
  POLYGON содержит:
    LINEARRING с 191 точкой данных
  POLYGON содержит:
    LINEARRING с 77 точками данных
  POLYGON содержит:
    LINEARRING с 152 точками данных
  POLYGON содержит:
    LINEARRING с 392 точками данных
  POLYGON содержит:
    LINEARRING с 93 точками данных
  POLYGON содержит:
    LINEARRING с 10105 точками данных
```

Как видите, шт. Калифорния состоит из 7 уникальных многоугольников, каждый из которых задан одним линейным кольцом. Это объясняется тем, что шт. Калифорния расположен на побережье и содержит шесть отдаленных островов, а также основную континентальную часть штата.

Закончим исследовать файл фигур со штатами США ответом на простой вопрос: каково расстояние от самой северной до самой южной точки в шт. Калифорния? На этот вопрос можно ответить по-разному, но на данный момент мы сделаем это вручную. Начнем с того, что определим в нем самые северные и самые южные точки:

```
import osgeo.ogr

def findPoints(geometry, results):
    for i in range(geometry.GetPointCount()):
        x,y,z = geometry.GetPoint(i)
        if results['north'] == None or results['north'][1] < y:
            results['north'] = (x,y)
        if results['south'] == None or results['south'][1] > y:
            results['south'] = (x,y)

    for i in range(geometry.GetGeometryCount()):
        findPoints(geometry.GetGeometryRef(i), results)

shapefile = osgeo.ogr.Open("tl_2014_us_state.shp")
layer = shapefile.GetLayer(0)
feature = layer.GetFeature(13)
geometry = feature.GetGeometryRef()

results = {'north' : None,
           'south' : None}

findPoints(geometry, results)

print("Самая северная точка: ({:.4f}, {:.4f})".format(
      results['north'][0], results['north'][1]))
print("Самая южная точка: ({:.4f}, {:.4f})".format(
      results['south'][0], results['south'][1]))
```

Функция findPoints() рекурсивно просматривает геометрию, извлекая индивидуальные точки и отождествляя их с самыми высокими и самыми низкими значениями переменной y (широты), которые затем заносятся в словарь результатов results, с тем чтобы главная программа могла ими воспользоваться.

Как видите, библиотека GDAL облегчает работу со структурой данных со сложной геометрией. Программа требует рекурсивной обработки, но она все еще тривиальна, по сравнению с попыткой прочесть данные непосредственно. Если вы выполните приведенную выше программу, то на экран будет выведен следующий результат:

```
Самая северная точка равна (-122.3782, 42.0095)
Самая южная точка равна (-117.2049, 32.5288)
```

Располагая этими двумя точками, теперь можно вычислить расстояние между ними. Как описано ранее, здесь мы должны рассчитать расстояние по дуге большого круга (по ортодромии), чтобы учесть кривизну земной поверхности. Мы сделаем это вручную, используя формулу гаверсинуса:

```
import math

lat1 = 42.0095
long1 = -122.3782

lat2 = 32.5288
```

```
long2 = -117.2049

rLat1 = math.radians(lat1)
rLong1 = math.radians(long1)
rLat2 = math.radians(lat2)
rLong2 = math.radians(long2)

dLat = rLat2 - rLat1
dLong = rLong2 - rLong1
a = math.sin(dLat/2)**2 + math.cos(rLat1) * math.cos(rLat2) \
                        * math.sin(dLong/2)**2
c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
distance = 6371 * c

print("Расстояние по дуге большого круга составляет {:0.0f} км." .format(distance))
```

Не переживайте по поводу используемых тут сложных математических формул; в сущности, мы конвертируем значения широты и долготы в радианы, вычисляем разность между двумя точками в значениях широты и долготы и затем, чтобы получить расстояние по дуге большого круга, пропускаем результаты через несколько тригонометрических функций. Значение 6371 – это радиус Земли в километрах.

Более подробная информация о формуле гаверсинуса находится на странице http://mathforum.org/library/drmath/view/51879.html.

Если выполнить эту программу, то компьютер сообщит расстояние от самой северной до самой южной точки в шт. Калифорния:

```
Расстояние по дуге большого круга составляет 1149 км.
```

Разумеется, есть и другие варианты расчета этого расстояния. Обычно формулу гаверсинуса не набирают непосредственно в программу, так как есть библиотеки, которые могут сделать это за вас. Но мы сознательно выполнили расчеты таким образом, чтобы показать, как это можно сделать.

Если вы захотите продолжить исследование этого вопроса, то можете попробовать написать программы, которые вычисляют следующее:

- самые восточные и самые западные точки шт. Калифорния;
- средняя точка шт. Калифорния;

Подсказка: долготу средней точки можно вычислить, взяв полусумму или среднее самых восточных и самых западных долгот.

- средняя точка шт. Аризона;
- расстояние между центром шт. Калифорния и центром шт. Аризона.

Как видите, ручная работа с данными ГИС не слишком обременительная. Несмотря на то что структуры данных и привлекаемый математический аппарат могут быть достаточно сложными и требуется разбираться в базовых понятиях ГИС, использование программных инструментов, таких как библиотека GDAL, делает данные доступными для понимания и простыми в работе.

Заключение

В этой главе мы обсудили многие базовые понятия, которые лежат в основе разработки геоинформационных систем (ГИС), исследовали несколько самых распространенных форматов данных ГИС и запачкались, экспериментируя с данными об американских штатах, скачанными с веб-сайта Бюро переписи населения США.

Мы увидели, что географические положения часто и далеко не всегда представлены координатами, узнали, что при вычислении расстояния между двумя точками требуется учитывать кривизну земной поверхности, и разобрались, почему всегда нужно знать об используемых в геопространственных данных единицах измерения.

Затем мы исследовали понятие картографических проекций, которые изображают трехмерную фигуру поверхности Земли как двумерную плоскость, и увидели, что имеются три основных класса картографических проекций: цилиндрические, конические и плоскостные. Мы обнаружили, что датумы представляют собой математические модели фигуры Земли, и узнали, что три наиболее распространенных действующих датума называются NAD 27, NAD 83 и WGS 84.

Затем мы исследовали понятие систем координат и увидели, что они используются для описания связи координат с конкретной точкой на поверхности Земли. Мы узнали, что географические (неспроецированные) системы координат представляют точки на поверхности Земли непосредственно, в то время как картографические (спроецированные) системы координат используют проекцию для того, чтобы представить Землю как двумерную координатную плоскость, на которую потом помещаются координаты.

Мы исследовали способы, которыми геоданные могут представлять геометрические фигуры в форме точек, ломаных линий и многоугольников, и рассмотрели несколько стандартных форматов данных ГИС, с которыми вы можете столкнуться. Мы увидели, что некоторые форматы данных работают с растровыми данными, в то время как другие используют данные в векторном формате.

Наконец, мы изучили варианты использования языка Python для выполнения различных геопространственных расчетов вручную на данных, загруженных из файлов фигур.

В следующей главе мы более подробно рассмотрим различные библиотеки, которыми можно воспользоваться для работы с геоданными.

Глава 3

Библиотеки Python для геопрограммирования

Эта глава посвящена разбору нескольких программных библиотек и других инструментов, которые могут использоваться для разработки геоприложений на языке Python. В частности, мы рассмотрим:

- библиотеки Python для чтения и записи геоданных;
- библиотеки Python для работы с картографическими проекциями;
- библиотеки для выполнения анализа геоданных и управления ими непосредственно из ваших программ на Python;
- инструменты для визуализации геоданных.

Отметим, что два типа геопространственных инструментов в этой главе не обсуждаются: геопространственные базы данных (ГБД) и геопространственный инструментарий для Интернета. Они будут подробно исследованы в этой книге позже.

Чтение и запись геоданных

В принципе, вы можете написать свой собственный синтаксический анализатор, который будет читать конкретный формат геоданных, вместе с тем намного легче для этих целей воспользоваться существующей библиотекой Python. Мы рассмотрим пакет, состоящий из двух популярных динамических библиотек для чтения и записи геоданных: **GDAL** и **OGR**.

Пакет GDAL/OGR

К сожалению, название этих двух библиотек зачастую сбивает с толку. **Библиотека единой абстрактной модели геоданных** `GDAL`, англ. аббревиатура для Geospatial Data Abstraction Library, первоначально была просто библиотекой для работы с растровыми геоданными, в то время как библиотека `OGR` предназначалась для работы с векторными данными. Однако теперь эти две библиотеки частично объединены, и их обычно скачивают и устанавливают вместе под объединенным названием `GDAL`. Чтобы избежать путаницы, условимся объединенную библиотеку

называть пакетом GDAL/OGR, а сокращение GDAL использовать исключительно для обозначения библиотеки трансляции растровых форматов геоданных.

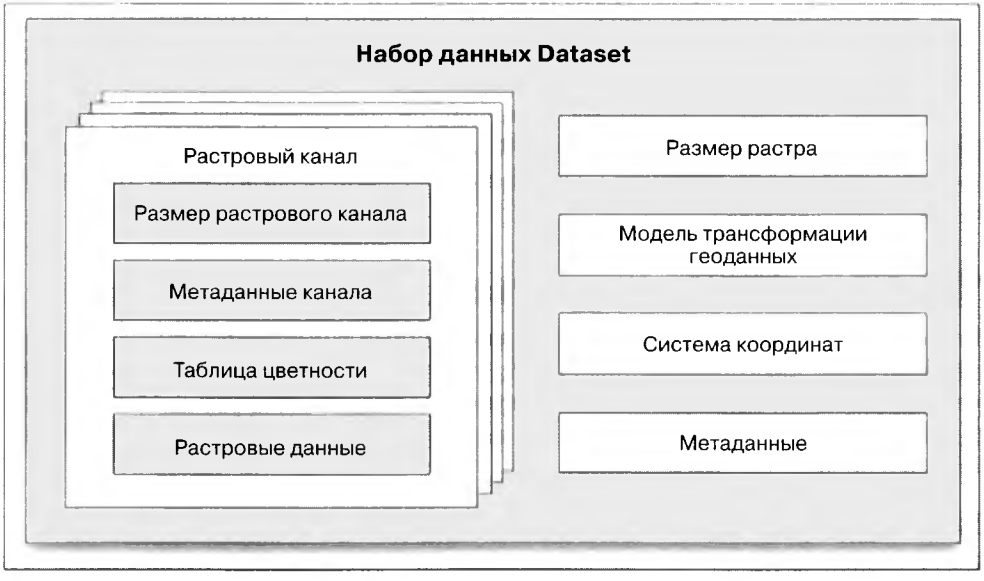
Стандартная инсталляционная версия библиотеки GDAL позволяет читать данные в 100 различных растровых форматах файлов и писать в 71 формате. Библиотека OGR по умолчанию поддерживает чтение данных в 42 различных векторных форматах файлов и запись в 39 форматах. Это делает пакет GDAL/OGR одним из самых мощных трансляторов геоданных из имеющихся и, конечно же, самой полезной общедоступной библиотекой для их чтения и записи.

Инсталляция пакета GDAL/OGR

Способы инсталляции пакета GDAL/OGR были разобраны в разделе «*Работа с данными ГИС вручную*» предыдущей главы. Для получения дополнительной информации по установке пакета GDAL/OGR на ваш компьютер обратитесь к инструкциям в указанном разделе.

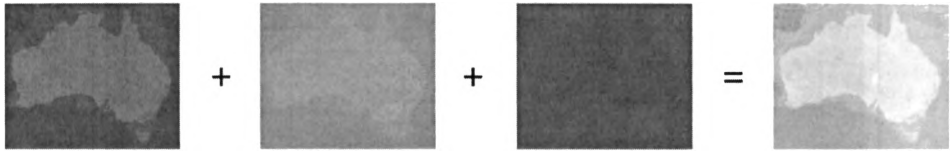
Концепция библиотеки GDAL

В библиотеке GDAL для описания растровых геопространственных данных используется следующая ниже модель данных:



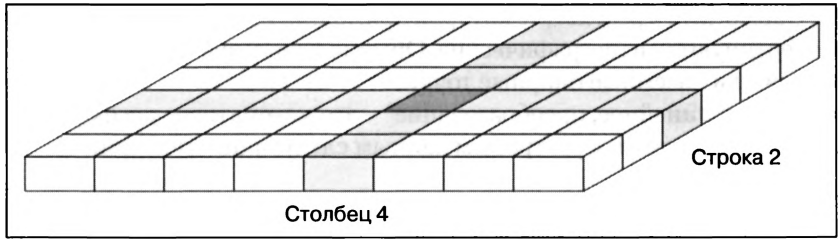
Для чтения и записи данных в растровом формате библиотека GDAL использует специальный объект **dataset**. Объект dataset является экземпляром класса gdal.Dataset и представляет один файл, в котором содержатся растровые данные.

Набор данных `dataset` разделен на несколько **каналов** (bands), каждый из которых содержит одну часть растровых данных. Например, растровый набор данных, формирующий изображение базовой карты, или подложки, может состоять из трех каналов, обозначая красный, зеленый и синий компоненты цвета полного изображения:



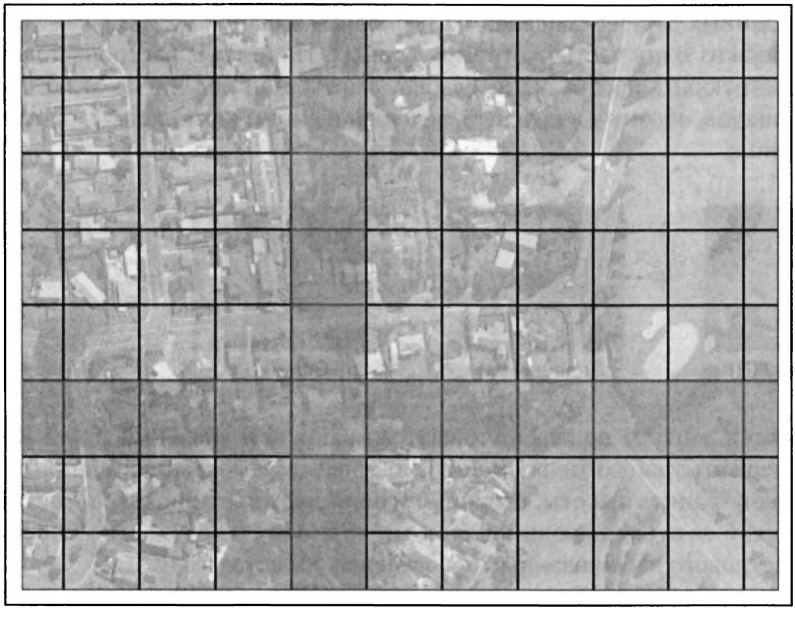
Разумеется, каналы не ограничиваются хранением значений цвета RGB; они могут содержать любую информацию по вашему усмотрению. Например, один канал может хранить высоты, второй – уровни влажности почвы, и третий – код, указывающий на существующий в этом месте тип растительности. Каждый канал внутри растрового набора данных представлен объектом `gdal.Band`.

Растровые данные каждого канала состоят из большого числа регулярных **ячеек**[1], организованных в строки и столбцы:



Размер растра содержит общую ширину изображения в пекселах и общую высоту изображения в строках. Каждая ячейка географически **привязывается** (georeference), то есть ей присваивается координата участка поверхности Земли:

[1] Ячейки, также *пиксели* или *точки* растра – это двумерные (площадные) объекты, являющиеся элементами регулярной прямоугольной решетки в модели данных, именуемой растром. – *Прим. перев.*



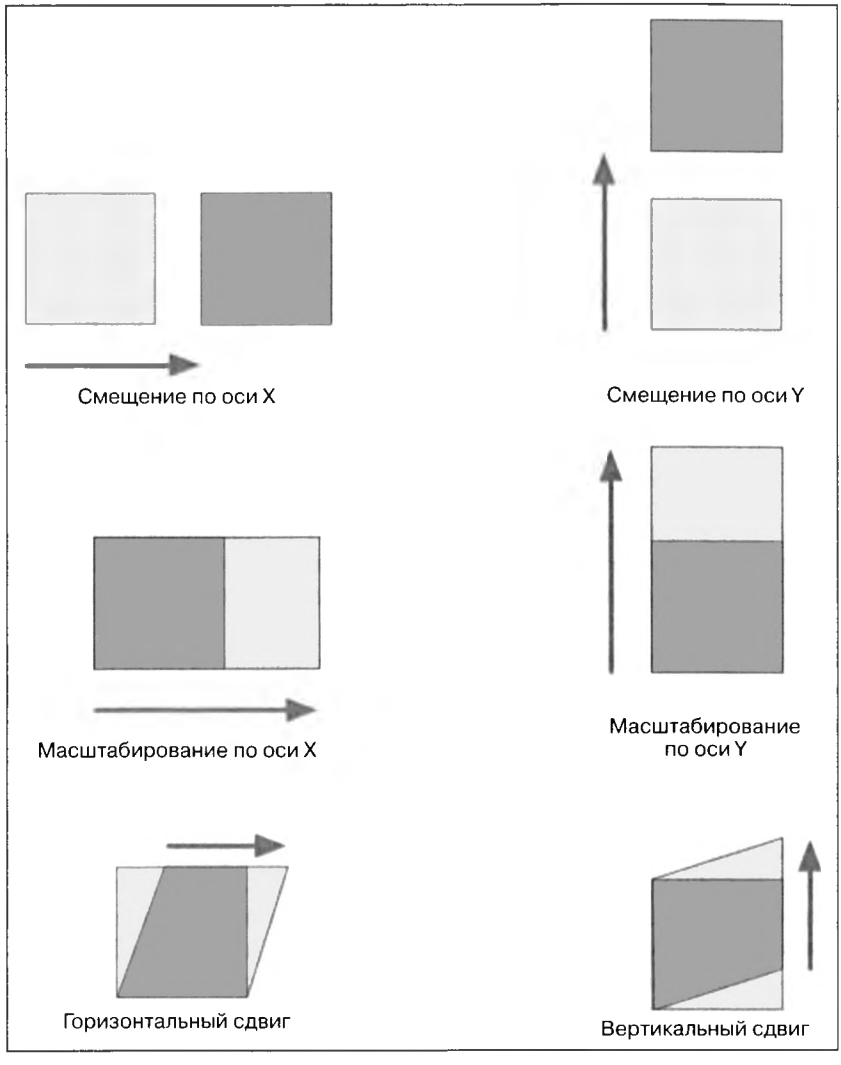
Геопривязка выполняется на основе выбранной **модели трансформации** (georeferencing transform)[1], в результате которой прямолинейные растровые координаты (x, y) конвертируются в географические. Обычно используют метод афинных преобразований или наземные опорные точки.

Афинное, или линейное, преобразование[2] – это математическая формула, которая позволяет применять к растровым данным следующие операции:

---

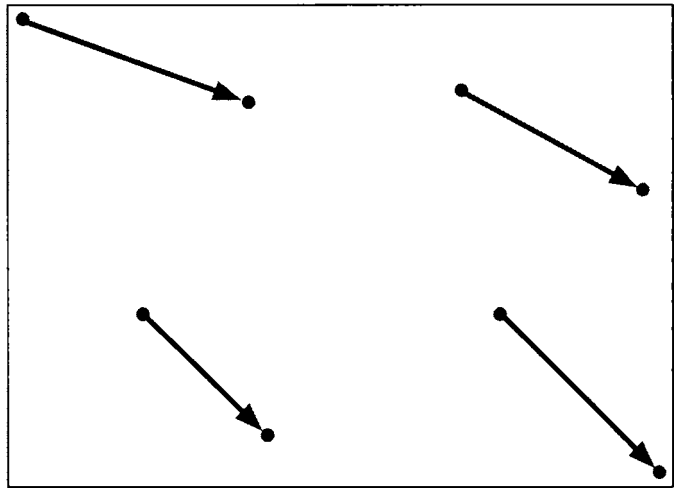
[1] Модель трансформации, или деформации, геоданных – математическая модель, которая преобразует ячейки растра геометрически. Трансформация растрового набора данных состоит в смещении растрового набора данных с его существующего положения в пространственно корректное. Чтобы для каждой ячейки в растре задать корректное положение прямоугольных координат, по выбору применяют полиномиальное, сплайновое или адаптивное преобразование. – *Прим. перев.*

[2] Аффинное преобразование, или полиномиальное преобразование, первого порядка – отображение плоскости в себя, при котором прямые в растровом наборе переходят в прямые в искривленном наборе. Используется для смещения, масштабирования и сдвига растрового набора данных, в результате чего квадраты ячеек преобразуются в параллелограммы любого размера и угловой направленности. – *Прим. перев.*



Наземные опорные точки помогают соотнести одну или несколько позиций внутри растра с соответствующими им геопривязанными координатами, как показано ниже[1]:

[1] Наземные опорные точки – это местоположения, которые однозначно идентифицируются в растровом наборе данных и в реальных координатах. – *Прим. перев.*



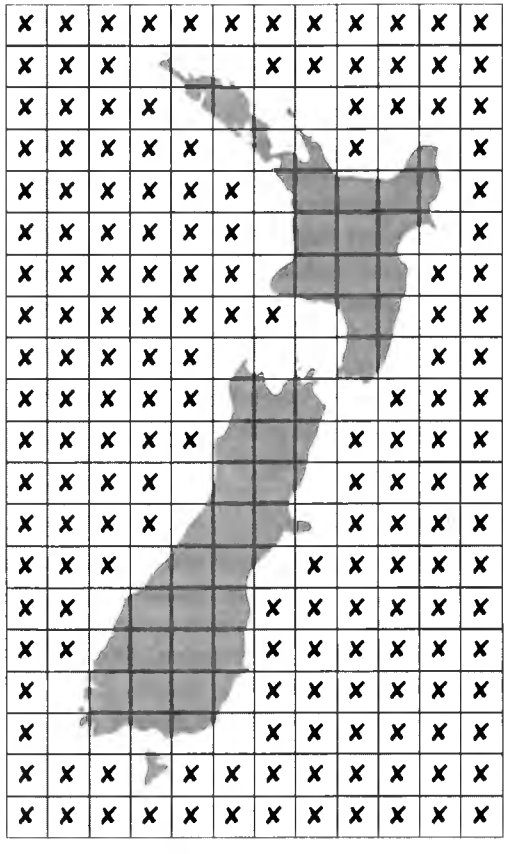
Отметим, что библиотека GDAL не переводит координаты при помощи наземных опорных точек – это является задачей конкретного приложения и связано со сложными математическими функциями, которые выполняют трансформацию.

Система координат описывает геопривязанные координаты, полученные в результате применения выбранного метода трансформации геоданных. Система координат включает в себя картографическую проекцию и геодезический датум, а также единицы измерения и масштаб, используемые в растровых данных. **Метаданные** содержат дополнительную информацию о наборе данных в целом.

Каждый растровый канал среди прочего содержит размер растрового канала и таблицу цветности. **Размер растрового канала** – это размер (число пикселов в ширину и линий в высоту) для данных внутри канала. Он может быть одинаковым с размером растра для всего набора данных, и тогда набор данных находится в полном разрешении, либо данные канала нуждаются в масштабировании, для того чтобы соответствовать набору данных dataset. **Таблица цветности** описывает то, каким образом значения пикселов транслируются в цвета.

Каждая ячейка канала растра содержит целое число или значение с плавающей точкой.

Поскольку охватываемая растровыми наборами данных земная поверхность имеет фигуру прямоугольника, то могут быть случаи, когда значения в некоторых ячейках отсутствуют. Например, набор данных с информацией по конкретной стране не будет содержать значений для ячеек, которые лежат за пределами этой страны, как показано на следующем ниже рисунке:



Каждый символ **X** на этом рисунке представляет ячейку, у которой значение отсутствует. Чтобы это стало возможным, в библиотеке GDAL поддерживается понятие **отсутствующих данных** (no data). Это специальное значение, которое хранится в ячейке, когда с ней никакое фактическое значение не связано.

Так как у каждого растрового канала обычно имеются тысячи (и даже миллионы) ячеек, то к ним обычно обращаются не поэлементно. Вместо этого для получения доступа ко всем данным канала *целиком* можно воспользоваться одним из двух возможных методов:

- можно воспользоваться методом `band.ReadRaster()`, чтобы прочитать часть или все данные канала в двоичную строку и затем для конвертирования этой строки в массив значений использовать встроенную библиотеку `struct`;
- можно воспользоваться методом `band.ReadArray()`, чтобы прочитать часть данных канала, сохраняя значения ячеек непосредственно в объект-массив библиотеки `NumPy`.

Оба подхода имеют свои преимущества и недостатки. Использование struct позволяет получать непосредственный доступ к значениям ячеек как к целым числам или числам с плавающей точкой, однако при этом за один раз можно легко извлечь всего одну строку данных. Подход на основе использования массива библиотеки NumPy облегчает одновременное чтение нескольких строк данных, но требует, чтобы была установлена сторонняя библиотека NumPy. Библиотека NumPy, кроме того, имеет недостаток, который заключается в том, что эта библиотека хранит каждое значение как собственный объект NumPy. Например, 16-разрядные целочисленные хранятся как объекты numpy.uint16; при необходимости библиотека NumPy автоматически конвертирует их в целые, однако эта операция может замедлить работу вашей программы, поэтому вам, возможно, придется выполнить эту конвертацию самостоятельно.

Отметим, что объект gdal.Band располагает соответствующими методами WriterRaster() и WriteArray(), позволяя использовать любой из подходов для записи растровых данных, а также их чтения.

Пример использования

Давайте рассмотрим использование библиотеки GDAL для записи растровых данных в файл формата GeoTIFF. Для нашего примера разделим всю поверхность Земли на 360 ячеек по горизонтали и 180 ячеек по вертикали, с тем чтобы каждая ячейка покрывала один градус широты и долготы. Для каждой ячейки сгенерируем одно случайное число между 1 и 100.

Начнем с создания непосредственно самого файла. Для этого сначала поручим библиотеке GDAL выбрать **драйвер**, который может создать нужный нам тип файла, и затем поручим драйверу создать файл:

```
from osgeo import gdal
driver = gdal.GetDriverByName("GTIFF")
dstFile = driver.Create("Образец растра.tiff", 360, 180, 1,
                        gdal.GDT_INT16)
```


Эта программа представлена по частям, с тем чтобы можно было объяснить все этапы работы; весь исходный код программы содержится в составе примеров программ, прилагаемых к данной главе. Ищите файл под названием writeRaster.py.

Метод gdal.Driver.Create() принимает следующие параметры:

- имя растрового файла;
- нужное число ячеек по горизонтали;
- нужное число ячеек по вертикали;
- нужное число каналов в файле;
- константа, задающая тип информации, хранимой в каждой ячейке. В нашем случае каждая ячейка будет содержать 16-разрядное целочисленное значение.

Наша следующая задача – задать **проекцию**, при помощи которой ячейки будут позиционироваться на поверхности Земли. В нашем случае мы воспользуемся да-

тумом WGS84[1]. Он позволит обращаться к позиции каждой ячейки при помощи значений широты и долготы:

```
from gdal import osr
spatialReference = osr.SpatialReference()
spatialReference.SetWellKnownGeogCS("WGS84")
dstFile.SetProjection(spatialReference.ExportToWkt())
```

Затем нам нужно задать модель **трансформации геоданных**, которая сообщает библиотеке GDAL, каким образом отображать каждую ячейку наших данных на земную поверхность. Модель трансформации геоданных задается списком из шести чисел в виде так называемой **матрицы аффинных преобразований**. К счастью, мы можем проигнорировать математику аффинных трансформаций и задать матрицу при помощи всего четырех значений:

- позиция X и Y верхнего левого угла верхней левой ячейки;
- ширина каждой ячейки, измеряемая в градусах долготы;
- высота каждой ячейки, измеряемая в градусах широты.

Ниже приведен фрагмент на Python, в котором задается трансформация геопривязки для нашего файла:

```
originX = -180
originY = 90
cellWidth = 1.0
cellHeight = 1.0

dstFile.SetGeoTransform([originX, cellWidth, 0,
                         originY, 0, -cellHeight])
```

Заодно получим ссылку на объект gdal.Band, который нужен для того, чтобы сохранить данные в одном-единственном растровом канале нашего файла:

```
band = dstFile.GetRasterBand(1)
```

Выполнив настройку растрового файла, теперь можно сгенерировать для него немного данных. Мы воспользуемся встроенным модулем random , чтобы создать список списков, каждый элемент которого содержит одно случайное число:

```
import random

values = []
for row in range(180):
    row_data = []
    for col in range(360):
        row_data.append(random.randint(1, 100))
    values.append(row_data)
```

---

[1] WGS 84 (англ. World Geodetic System 1984) – всемирная система геодезических параметров Земли 1984 года, в число которых входит система геоцентрических координат. В отличие от локальных систем, является единой системой для всей планеты. – *По материалам Википедии.*

Чтобы сохранить эти значения в файле при помощи встроенного модуля struct, нам придется писать их на диск по одной строке на каждой итерации цикла. Ниже приведен фрагмент программы, который показывает, как это сделать:

```
import struct

fmt = "<" + ("h" * band.XSize)

for row in range(180):
    scanline = struct.pack(fmt, *values[row])
    band.WriteRaster(0, row, 360, 1, scanline)
```

Как вариант, если у вас установлена библиотека NumPy, целиком весь массив values можно записать в файл как массив NumPy:

```
import numpy

array = numpy.array(values, dtype=numpy.int16)
band.WriteArray(array)
```

На этом работа программы завершена. Если ее выполнить, то будет создан файл под названием Образец растра.tiff, который содержит 360×180 случайных чисел. Теперь напишем другую программу, которая будет читать содержимое этого файла.

К счастью, чтение растровых данных проще, чем их запись. Начнем с открытия набора данных и получения ссылки на наш (один-единственный) растровый канал:

```
from osgeo import gdal

srcFile = gdal.Open("Образец растра.tiff")
band = srcFile.GetRasterBand(1)
```

Эта программа под названием readRaster.py включена в состав примеров программ, прилагаемых к данной главе, которые можно скачать отдельно.

Теперь мы готовы прочесть содержимое файла. Вот как это делается при помощи встроенного модуля struct:

```
import struct

fmt = "<" + ("h" * band.XSize) # длина строки канала = 180

for row in range(band.YSize):
    scanline = band.ReadRaster(0, row, band.XSize, 1,
                               band.XSize, 1,
                               band.DataType)
    row_data = struct.unpack(fmt, scanline)
    print(row_data)
```

Не переживайте насчет параметров метода ReadRaster; так как этот метод при считывании может данные масштабировать, мы должны дважды предоставить

число ячеек по горизонтали и по вертикали, а также позицию начальной ячейки. Если вы хотите узнать подробности, то можете найти определение этого метода в документации GDAL.

Как видите, мы просто распечатываем возвращаемые значения ячеек для их просмотра. Проделаем то же самое снова, но на этот раз при помощи библиотеки NumPy:

```
values = band.ReadAsArray()
for row in range(band.XSize - 180): # отсчет с нуля
    print(values[row])
```

Отметим, что метод ReadAsArray() доступен, только если библиотека NumPy установлена.

Как видите, после того как вы преодолели сложности конвертирования исходных двоичных данных на Python, чтение растровых данных выполняется довольно прямолинейно, будь то при помощи библиотеки GDAL на основе встроенной библиотеки struct либо при помощи сторонней библиотеки NumPy.

Концепция библиотеки OGR

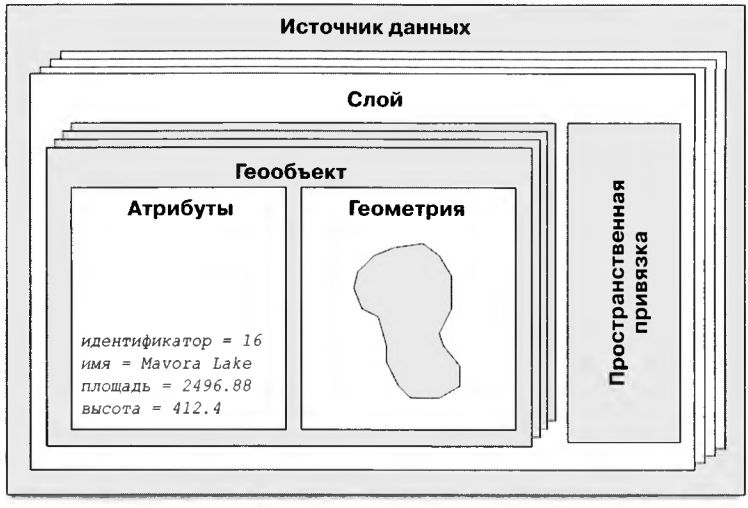
Как мы уже убедились, библиотека OGR используется для чтения и записи геоданных в векторном формате. В библиотеке OGR используется термин **источник данных**, который представляет файл или другой источник векторных данных. Источник данных состоит из одного или более тематических **слоев**, каждый из которых представляет набор данных с сопутствующей информацией. Многие источники данных состоят всего из одного слоя, вместе с тем некоторые источники данных могут быть более сложными. Например, один составной источник данных может хранить контуры территории, дорожную сеть, городские границы и местоположения больниц, причем все будут представлены отдельным слоем внутри источника данных.

Каждый слой состоит из списка **геообъектов**. Каждый такой геообъект описывает одну сущность реального мира. Например, в слое «городов» каждый геообъект представляет один город.

Каждый геообъект слоя имеет **геометрию**, то есть фактический геопространственный элемент данных, представляющий фигуру геообъекта или его положение в пространстве, а также список **атрибутов**, которые обеспечивают дополнительную метаинформацию о геообъекте.

Наконец, помимо самих геообъектов, в каждом слое имеется **пространственная привязка** (spatial reference), задающая используемые в геообъектах слоя проекцию и датум.

Следующая ниже иллюстрация показывает, каким образом все эти элементы друг с другом связаны:



Хотя эта структура может показаться достаточно сложной, универсальность такого подхода позволяет библиотеке OGR работать со всеми видами векторных форматов данных. Например, в формате файла фигур многослойность не поддерживается, и в библиотеке OGR с учетом этого используется сообразная модель, которая представляет файл фигур как источник данных, состоящий всего из одного слоя.

Пример использования

Учитывая, что в предыдущей главе вы научились применять библиотеку `OGR` для чтения данных файла фигур, теперь приглядимся поближе к процедуре записи векторных данных при помощи этой библиотеки. Мы напишем простую программу на Python, которая генерирует набор произвольных геометрий точки и сохраняет их в файле фигур вместе с некоторой атрибутивной информацией.

Начнем с импорта необходимых модулей и создания каталога, где будет содержаться сгенерированный файл фигур:

```
import os, os.path, shutil, random
from osgeo import ogr, osr

if os.path.exists("Тестовый_файл_фигур"):
    shutil.rmtree("Тестовый_файл_фигур")
os.mkdir("Тестовый_файл_фигур")
```

Весь исходный код этой программы содержится в файле `writeVector.py` среди файлов примеров, прилагаемых к данной главе.

Отметим, что мы удаляем каталог нашего файла фигур, в случае если он уже существует. Это позволит выполнять программу многократно, не заставляя вас каждый раз заботиться об удалении файла фигур.

Теперь можно назначить наш файл фигур в качестве источника данных библиотеки OGR. Как и с библиотекой GDAL, мы сначала выбираем **драйвер**, который соответствует типу источника данных, который мы хотим создать, и затем поручаем драйверу создать источник данных. Ниже приведен соответствующий фрагмент программы:

```
driver = ogr.GetDriverByName("ESRI Shapefile")
path = os.path.join("Тестовый_файл_фигур", "shapefile.shp")
datasource = driver.CreateDataSource(path)
```

Затем нам нужно определить слой. Для этого мы сначала определяем пространственную привязку для использования в наших данных:

```
spatialReference = osr.SpatialReference()
spatialReference.SetWellKnownGeogCS('WGS84')
```

И затем можно создать непосредственно сам слой:

```
layer = datasource.CreateLayer("layer", spatialReference)
```

Учитывая, что мы собираемся сохранить в файле атрибутивную информацию, после этого нам нужно сообщить библиотеке OGR, какие атрибуты будут использоваться и какой тип данных будет храниться. Эта информация добавляется непосредственно в слой карты при помощи класса `ogr.FieldDefn`:

```
field = ogr.FieldDefn("ID", ogr.OFTInteger)
field.SetWidth(4)
layer.CreateField(field)

field = ogr.FieldDefn("NAME", ogr.OFTString)
field.SetWidth(20)
layer.CreateField(field)
```

Создав файл фигур, теперь сохраним в нем немного информации. Мы сгенерируем 100 геометрий точки и вычислим идентификатор и имя, которые свяжем с каждым геообъектом. Для этого мы воспользуемся встроенным модулем `random`:

```
for i in range(100):
    id = 1000 + i
    lat = random.uniform(-90, +90)
    long = random.uniform(-180, +180)
    name = "point-{}".format(i)
    wkt = "POINT({} {})".format(long, lat)
    geometry = ogr.CreateGeometryFromWkt(wkt)
```

Как видите, мы определяем геометрию точки как строку в формате WKT, которую затем конвертируем в объект `ogr.Geometry`. Затем для каждой точки нам нужно создать объект `ogr.Feature`, обозначающий пространственный объект, и сохранить значения геометрии и атрибутов в этом объекте:

```
    feature = ogr.Feature(layer.GetLayerDefn())
    feature.SetGeometry(geometry) # задать геометрию объекта
```

```
    feature.SetField("ID", id)  # установить значения его атрибутов
    feature.SetField("NAME", name)
```

Наконец, нужно разместить этот объект в слое, чтобы его можно было сохранить в файле:

```
    layer.CreateFeature(feature)
    feature.Destroy()  # уничтожить объект, освободив ресурсы
datasource.Destroy() # уничтожить источник данных, освободив ресурсы
```

На этом работа программы завершена. Если ее выполнить, то будет создана папка с именем Тестовый_файл_фигур, в которой будет содержаться несколько файлов с данными файла фигур.

Чтобы прочесть содержимое сгенерированного файла фигур, воспользуемся тем же самым приемом, который мы рассматривали в предыдущей главе. Ниже приведена простая программа, которая снова загрузит геометрии точки в память и выведет на экран сохраненные ранее координаты и атрибуты:

```
from osgeo import ogr

shapefile = ogr.Open("Тестовый_файл_фигур/shapefile.shp")
layer = shapefile.GetLayer(0)

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    id = feature.GetField("ID")
    name = feature.GetField("NAME")
    geometry = feature.GetGeometryRef()
    print(i, name, geometry.GetX(), geometry.GetY())

shapefile = None
```

Эта программа под названием readVector.py находится среди других примеров программ, прилагаемых к данной главе.

Как видите, использование библиотеки OGR для работы с источником векторных данных не представляет особого труда.

Документация по GDAL/OGR

Динамические библиотеки GDAL и OGR хорошо задокументированы, но есть одна загвоздка для программистов на Python. Эти библиотеки и связанные с ними инструменты командной строки написаны на C и C++. Имеющиеся для этих библиотек привязки позволяют обращаться к ним из ряда других языков, включая Python, однако вся документация по ним написана для версии библиотек, предназначенных для C++. Эта особенность может потребовать значительных усилий во время ее чтения – дело в том, что не только все сигнатуры методов написаны на C++, но и в привязках Python были заменены многие имена, чтобы придать методам и классам более «питоновский» вид.

К счастью, библиотеки Python в основном самодокументированы благодаря строковым литералам docstring, встроенным в сами привязки Python. То есть можно исследовать документацию при помощи таких инструментов, как встроен-

ная утилита Python pydoc, которую можно запускать на выполнение из командной строки, как показано ниже:

```
pydoc -g osgeo
```

В результате откроется окно приложения, позволяющее вам читать документацию при помощи веб-браузера. Как вариант, если вы хотите узнать об отдельном методе или классе, можно воспользоваться встроенной командой Python help(), применяемой из командной строки Python, как показано ниже:

```
>>> from osgeo import ogr
>>> help(ogr.DataSource.CopyLayer)
```

Не все методы задокументированы, поэтому для получения дополнительной информации на веб-сайте библиотеки GDAL вам, возможно, придется обратиться к документации для C++, при этом некоторые строки docstring, которые там есть, скопированы непосредственно из документации для C++ – впрочем, документация по GDAL/OGR в целом превосходна и должна позволить вам прибавить скорости при освоении этого пакета.

Работа с проекциями

Одна из трудностей работы с геоданными состоит в том, что геодезические положения (точки на земной поверхности) часто проецируются на двумерную координатную плоскость при помощи картографической проекции. Мы уже рассмотрели картографические проекции в предыдущей главе и знаем, что, имея некие геоданные, всегда нужно знать, какая проекция в них используется. Кроме того, необходимо знать, какой датум (то есть математическая модель фигуры Земли) в них взят за основу.

Но чаще всего при работе с геоданными трудность вызывает то, что приходится переводить данные из одной проекции или датума в другую. К счастью, существует инструмент, который выполняет эту задачу просто и понятно. Это библиотека Python pyproj.

Библиотека pyproj

Библиотека Python pyproj – это обертка вокруг динамической библиотеки **PROJ.4**, написанной на C. Название последней обозначает, что это версия 4 библиотеки PROJ. Библиотека PROJ первоначально была написана Геологической службой США, англ. название US Geological Survey (USGS), для работы с картографическими проекциями и уже много лет широко используется в геопространственном программном обеспечении. Библиотека pyproj позволяет получать доступ к функциональности PROJ.4 из ваших программ на Python.

Инсталляция библиотеки

Библиотека pyproj предлагается для Windows, Mac OS X и любой операционной системы на основе POSIX. Главная страница библиотеки находится на https://github.com/jswhit/pyproj.

Способ инсталляции библиотеки pyproj зависит от операционной системы, установленной на вашем компьютере:

- в Windows сначала необходимо установить базовую динамическую библиотеку PROJ.4. Ее двоичный установщик находится на https://github.com/OSGeo/proj.4/wiki. После инсталляции библиотеки PROJ.4 нужно установить библиотеку Python pyproj. Питоновский wheel-файл (.whl)[1] для библиотеки pyproj, который работает с вашей версией Python, можно скачать с http://www.lfd.uci.edu/~gohlke/pythonlibs/#pyproj. После того как вы скачали соответствующий wheel-файл, можно его установить при помощи диспетчера пакетов Python pip. Исчерпывающие инструкции по установке дистрибутивных пакетов wheel, используя для этого диспетчер пакетов Python pip, находятся на странице https://pip.pypa.io/en/latest/user_guide.html#installing-from-wheels;

После извлечения пакета с базовой динамической библиотекой из архива нужно открыть окно командной строки и прописать месторасположение папки в системном пути. Например:

```
C:\> set PATH=%PATH%;C:\PROJ\BIN
```

Если архив извлечен не в корневом каталоге C:\, то для того, чтобы библиотека находила файлы инициализации, также следует установить переменную среды окружения PROJ_LIB. Например:

```
C:\> set PROJ_LIB=C:\<ПУТЬ>\PROJ\NAD
```

То же самое можно сделать непосредственно в окне **Переменные среды** (**Панель управления** ⇨ **Система и безопасность** ⇨ **Система** ⇨ **Дополнительные параметры системы** ⇨ **Переменные среды** ⇨ **Системные переменные**). После этих настроек можно прямо из командной строки проверить работу различных утилит PROJ. Например:

```
C:\> proj.exe  -I +proj=utm +zone=11 +ellps=WGS84
332000 4000000                    (ввести данные)
118d52'1.182"W  36d7'48.593"N     (результат)
<Ctrl+Z>
C:\>
```

На странице Лаборатории флюоресцентной динамики Калифорнийского университета, ссылка на которую приведена выше, имеется несколько вариантов wheel-файлов для установки пакета Python pyproj, предназначенных для работы с Python 3.5. При тестировании использовался pyproj-1.9.5-cp35-none-win_amd64.whl.

- в Mac OS X можно установить pyproj вместе с базовой динамической библиотекой PROJ.4, просто набрав в командной строке pip install pyproj;

---

[1] Wheel-файл – это файл в формате zip с дистрибутивным (собранным) пакетом Python. – *Прим. перев.*

Если эта команда не работает, то можно установить версию платформы PROJ.4 с http://www.kyngchaos.com/software/frameworks. После ее установки диспетчер пакетов pip должен успешно установить библиотеку pyproj.

- в Linux динамическую библиотеку PROJ.4 можно установить, воспользовавшись вашим любимым диспетчером пакетов, и затем установить саму библиотеку pyproj, используя диспетчер пакетов Python pip. Как вариант можно загрузить исходный код pyproj, скомпилировать его и установить библиотеку самостоятельно. Например, в простейшем случае:

```
sudo dnf install python3-pyproj
```

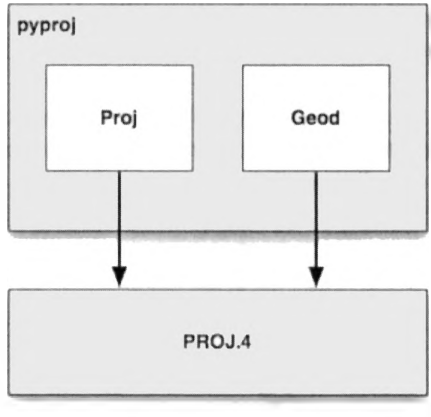
Чтобы проверить, что инсталляция прошла успешно, запустите командную строку Python и попробуйте набрать следующее:

```
import pyproj
print(pyproj.__version__)
```

Если эти команды работают без ошибки, то, значит, инсталляция библиотеки pyproj на ваш компьютер прошла успешно.

Концепция библиотеки

Библиотека pyproj состоит всего из двух классов: Proj и Geod.



Класс Proj выполняет преобразования значений долготы и широты в прямоугольные координаты (x, y) и наоборот, тогда как класс Geod выполняет различные расчеты на дуге большого круга и углов. Оба реализованы поверх динамической библиотеки PROJ.4. Приглядимся к этим двум классам поближе.

Класс Proj .

Proj – это класс картографической трансформации, который позволяет переводить географические координаты (то есть значения широты и долготы) в пря-

моугольные координаты на карте и наоборот (значения (x, y) по умолчанию исчисляются в метрах).

При создании нового экземпляра `Proj` указываются проекция, датум и другие значения, при помощи которых описывается метод трансформирования координат. Например, чтобы указать поперечную проекцию Меркатора и эллипсоид WGS84, используется следующий ниже фрагмент программного кода на Python:

```
projection = pyproj.Proj(proj='tmerc', ellps='WGS84')
```

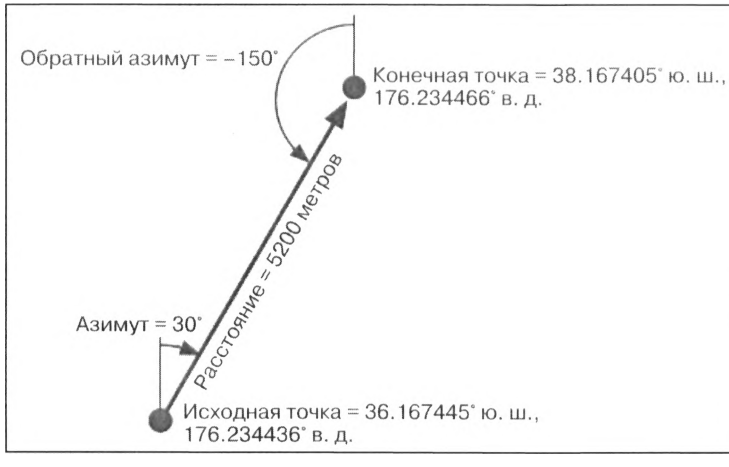
После создания экземпляра `Proj` его можно применить для перевода широты и долготы в прямоугольные координаты (x, y) заданной проекции. Его также можно применить для выполнения *обратной* проекции, то есть перевода из координаты (x, y) назад в значение широты и долготы.

Полезная функция `transform()` может применяться для непосредственного перевода координат из одной проекции в другую. Вы просто передаете исходные координаты, объект `Proj`, который описывает *проекцию* исходных координат и нужную конечную проекцию. Это может быть очень удобно как при однократном, так и при *массовом* переводе координат.

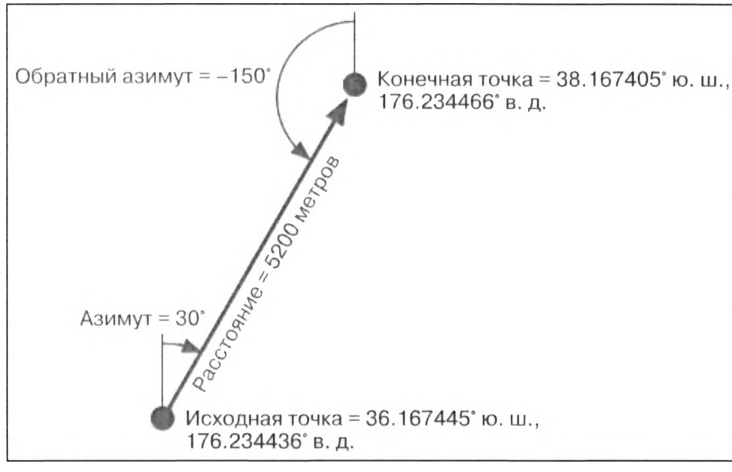
Класс Geod

`Geod` – это класс геодезических вычислений. Он позволяет выполнять различные расчеты на дуге большого круга. Мы уже касались расчетов на дуге большого круга, когда рассматривали методы точного вычисления расстояния между двумя точками на поверхности Земли. Однако класс `Geod` может делать гораздо больше, чем только это:

- метод `fwd()` берет исходную точку, азимут (угловое направление) и расстояние и возвращает конечную точку и обратный азимут (угол от конечной точки назад к исходной точке):



- метод `inv()` берет две координаты и возвращает прямой и обратный азимуты, а также расстояние между ними:



- метод `npts()` вычисляет координаты для ряда точек, расположенных с равноудаленными интервалами вдоль геодезической линии, идущей от исходной точки к конечной:

При создании нового объекта `Geod` указывают эллипсоид, который используется при выполнении геодезических расчетов. Эллипсоид можно выбрать из нескольких предварительно заданных эллипсоидов, либо можно ввести параметры эллипсоида непосредственно (экваториальный радиус, полярный радиус и т. д.).

Пример использования

Следующий ниже пример начинается с указания географического положения в координатах 17-й зоны проекции Меркатора (UTM) и затем при помощи двух

объектов Proj, один с заданной 17-й зоной UTM и другой с координатной проекцией (широта и долгота), выполняет перевод прямоугольных координат этого географического положения в значения широты и долготы:

```
import pyproj

UTM_X = 565718.523517
UTM_Y = 3980998.9244

srcProj = pyproj.Proj(proj="utm", zone="17",
                      ellps="clrk66", units="m")
dstProj = pyproj.Proj(proj='longlat', ellps='WGS84',
                      datum='WGS84')
long,lat = pyproj.transform(srcProj, dstProj, UTM_X, UTM_Y)

print("Координата 17-й зоны UTM " +
      "({:.4f}, {:.4f}) ".format(UTM_X, UTM_Y) +
      "= {:.4f}, {:.4f}".format(long, lat))
```

Второй пример принимает эти расчетные значения широты и долготы и при помощи объекта Geod вычисляет еще одну точку в 10 км к северо-востоку от этого географического положения:

```
angle = 315 # 315 градусов = северо-восток.
distance = 10000
geod = pyproj.Geod(ellps='clrk66')
long2,lat2,invAngle = geod.fwd(long, lat, angle, distance)

print("Координата {:.4f}, {:.4f}".format(lat2, long2) +
      " находится в 10 км к северо-востоку от " +
      "{:.4f}, {:.4f}".format(lat, long))
```

Оба приведенных примера находятся в файле pyproj_example.py среди примеров исходного кода, прилагаемых к данной главе.

Документация

В принципе, качество документации по библиотеке pyproj на веб-сайте библиотеки и в файле README, поставляемом вместе с исходным кодом, превосходное. В ней описывается, как использовать классы и методы, что они делают и какие параметры требуются. Однако документация довольно скупа, когда дело доходит до параметров, используемых при создании нового объекта Proj. Вот что там говорится:

> *«Экземпляр класса Proj инициализируется контрольным параметром картографической проекции proj в виде пар ключ-значение. Пары ключ-значение могут быть переданы в словаре либо как именованные аргументы, либо в виде строкового значения proj4 (совместимого с командой proj)».*

Документация действительно предоставляет ссылку на веб-сайт, где приводится перечень стандартных картографических проекций и связанных с ними параметров, но, чтобы понять, что именно эти параметры означают, как правило, от вас потребуется углубиться непосредственно в документацию по динамической биб-

лиотеке PROJ, которая, к сожалению, невразумительная и запутанная, тем более что главное руководство написано для PROJ версии 3, а дополнения к нему – для версий 4 и 4.3, и, чтобы попытаться понять все это, могут потребоваться значительные усилия.

К счастью, в большинстве случаев вам вообще не потребуется обращаться к документации по PROJ. Работая с геоданными при помощи библиотек `GDAL` или `OGR`, вы можете легко извлекать проекцию как «строку proj4», которую затем можно передавать непосредственно в функцию инициализации `Proj`. Если вы хотите «зашить» значение проекции в программный код, то, в общем-то, можно выбрать проекцию и эллипсоид, используя, соответственно, параметры proj = «...» и ellps = «...». Если же вы хотите добиться чего-то большего, чем это, то вам придется обратиться к документации по PROJ для получения дополнительной информации.

Чтобы узнать о PROJ больше и почитать исходную документацию, обратитесь к вики-книге на странице https://github.com/OSGeo/proj.4/wiki, где можно найти все, что вам нужно.

Геоанализ и геообработка

Учитывая, что геоданные связаны с геометрическими фигурами – точками, линиями и многоугольниками, часто приходится выполнять различные расчеты с их участием. К счастью, для выполнения именно таких задач существует несколько очень мощных инструментов. По причинам, которые мы опишем чуть позже, наиболее предпочтительной для решения подобного рода вычислительно-геометрических задач на Python является библиотека **Shapely**.

Библиотека Shapely

`Shapely` – это библиотека Python для управления двумерными геопространственными геометриями и выполнения их анализа. Она основана на динамической библиотеке GEOS (движке, используемом в PostGIS), в которой реализован широкий круг операций, связанных с управлением геоданными на C++. Библиотека GEOS, в свою очередь, основана на другой библиотеке, написанной на Java, под названием Java Topology Suite (JTS), то есть пакете Java с топологическими функциями, который предлагает ту же самую функциональность программистам на Java. Библиотека `Shapely`, являясь оберткой вокруг динамической библиотеки GEOS, предоставляет питоновский интерфейс, который облегчает использование ее функционала прямо из ваших программ на Python.

Инсталляция библиотеки

Библиотека `Shapely` работает во всех основных операционных системах, включая Windows, Mac OS X и Linux. Главный веб-сайт библиотеки находится на https://github.com/Toblerity/Shapely. Раздел каталога библиотек Python, посвященный `Shapely`, тоже содержит множество полезной информации, и она находится на https://pypi.python.org/pypi/Shapely.

Способ установки библиотеки Shapely зависит от того, какая операционная система используется на вашем компьютере:

- в Windows можно установить предварительно созданный питоновский wheel-файл, который содержит библиотеку Shapely, а также базовую динамическую библиотеку GEOS. Для этого пройдите на http://www.lfd.uci.edu/~gohlke/pythonlibs/#shapely и скачайте соответствующий wheel-файл (.whl) для версии Python, которую вы используете. Затем следуйте инструкциям, приведенным на https://pip.pypa.io/en/latest/user_guide.html#installing-from-wheels, чтобы установить wheel-файл на компьютер;
- в Mac OS X, прежде чем устанавливать библиотеку Shapely, сначала необходимо установить динамическую библиотеку GEOS. Для этого пройдите на http://www.kyngchaos.com/software/frameworks, скачайте и установите платформу GEOS. После этого можно воспользоваться диспетчером пакетов Python pip, который установит саму библиотеку Shapely. Чтобы та распознала библиотеку GEOS, следует установить Shapely, набрав в окне терминала следующий текст:

```
export LDFLAGS=`/Library/Frameworks/GEOS.framework/Versions/3/
unix/bin/geos-config --libs`
export CFLAGS=`/Library/Frameworks/GEOS.framework/Versions/3/unix/
bin/geos-config --cflags`

pip install shapely
```

Может быть выдано предупреждение о неправильной версии файла, однако его можно проигнорировать, поскольку этот факт не помешает библиотеке Shapely в работе.

- на компьютере на базе Fedora 24/Linux сначала нужно установить динамическую библиотеку GEOS, воспользовшись для этого диспетчером пакетов вашего компьютера (или в окне терминала sudo dnf install geos geos-devel), и затем установить саму библиотеку Python Shapely. Если требуется дополнительная информация об установке динамической библиотеки GEOS, то веб-сайт библиотеки находится на http://trac.osgeo.org/geos. В простейшем случае команды будут следующими:

```
dnf install geos geos-devel
dnf install python3-shapely
sudo pip3 install shapely --upgrade
```

После установки библиотеки Shapely запустите интерпретатор Python и попробуйте набрать нижеследующие операторы:

```
import shapely
import shapely.speedups
print(shapely.__version__)
print(shapely.speedups.available)
```

В результате будет напечатан номер версии установленной библиотеки и сообщено о наличии модуля shapely.speedups. Если модуль имеется, то для выполнения всей тяжелой работы библиотека Shapely будет пользоваться базовой динамической библиотекой GEOS; без модуля speedups библиотека Shapely главным образом продолжит функционировать, но при этом будет выполняться очень медленно.

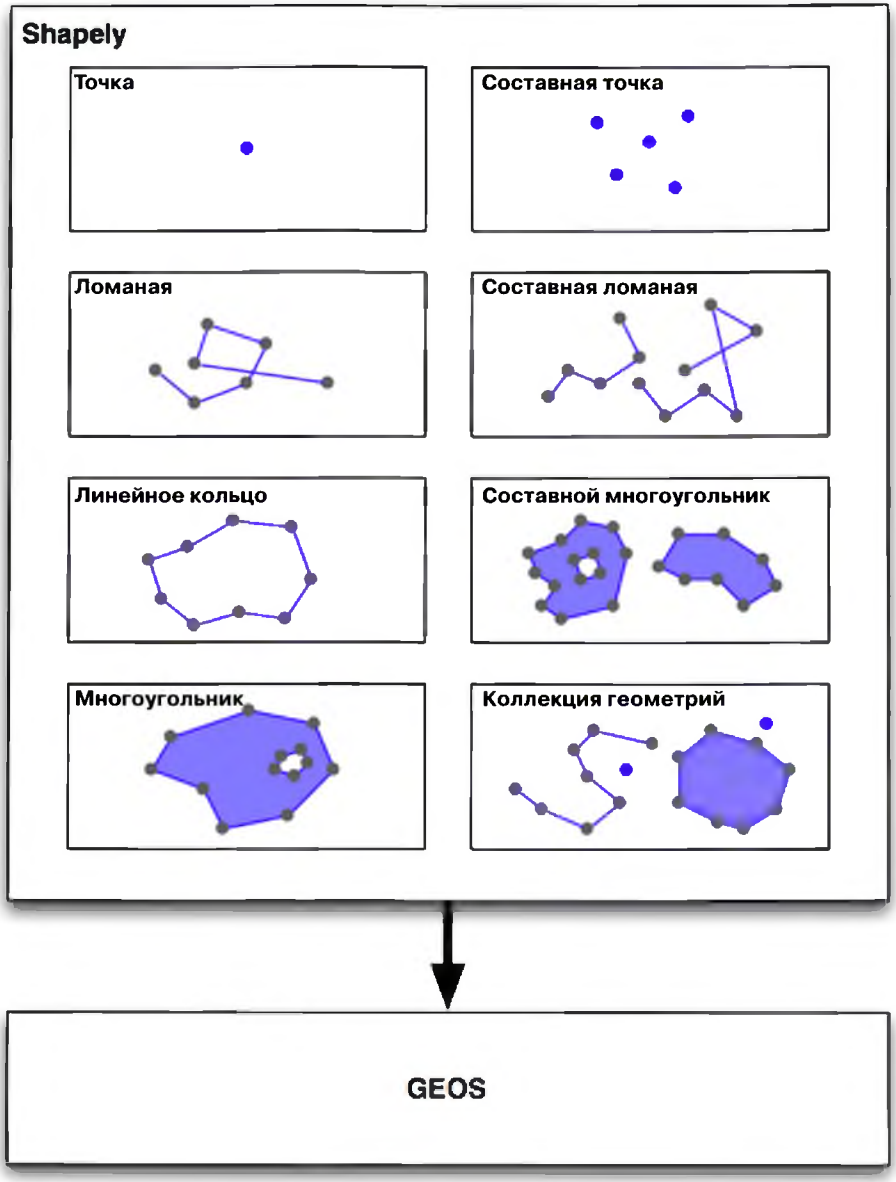
Концепция библиотеки

Библиотека Shapely разделена на ряд отдельных модулей, которые импортируются по мере необходимости. Наиболее часто используемые модули следующие:

- shapely.geometry: задает все используемые в Shapely классы базовых геометрических фигур;
- shapely.wkt: предоставляет функции конвертирования объектов геометрии Shapely в строки в формате стандартного текстового представления WKT и наоборот;
- shapely.wkb: предоставляет функции конвертирования объектов геометрии Shapely в двоичные данные в формате стандартного двоичного представления WKB и наоборот;
- shapely.ops: предоставляет функции для выполнения пакетных операций на ряде объектов геометрии.

При том что вы почти всегда будете импортировать в свою программу модуль shapely.geometry, импорт других модулей не обязателен и выполняется, только если в этом есть необходимость.

Приглядимся поближе к объектам геометрии, определенным в модуле shapely.geometry. Библиотека Shapely поддерживает 8 фундаментальных типов геометрий, или геометрических примитивов:



Каждый из этих типов геометрий реализован как класс внутри модуля `shapely.geometry`:

- `shapely.geometry.Point` (точка) изображает отдельную точку в пространстве. Точки могут быть двумерными (x, y) или трехмерными (x, y, z);

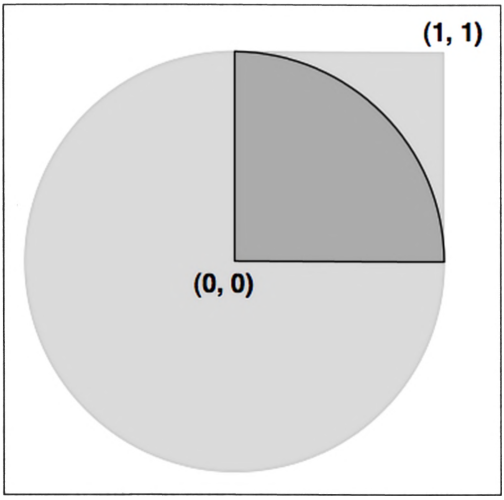
- `shapely.geometry.LineString` (полилиния) изображает ломаную, то есть последовательность точек, объединенных в линию. Ломаные линии могут быть простыми (без пересекающихся отрезков) или сложными (где два отрезка ломаной самопересекаются);
- `shapely.geometry.LinearRing` (линейное кольцо) изображает ломаную линию, где первая и последняя координаты совпадают. Отрезки внутри линейного кольца не могут самопересекаться или касаться друг друга;
- `shapely.geometry.Polygon` (многоугольник) изображает заполненную область, внутри которой дополнительно может быть одна или несколько «дыр»;
- `shapely.geometry.MultiPoint` (мультиточка) изображает составную точку, то есть коллекцию точек;
- `shapely.geometry.MultiLineString` (мультиполилиния) изображает составную ломаную, то есть коллекцию ломаных линий;
- `shapely.geometry.MultiPolygon` (мультимногоугольник) изображает составной многоугольник, то есть коллекцию многоугольников;
- `shapely.geometry.GeometryCollection` (коллекция геометрий) представляет коллекцию, состоящую из любой комбинации точек, линий, линейных колец и многоугольников.

Помимо того что библиотека `Shapely` может представлять эти разные типы геометрий, она также предлагает большое количество методов и атрибутов для управления ими и выполнения их анализа. Например, класс `LineString`, описывающий ломаную, содержит атрибут `length`, равный длине всех отрезков, которые ее составляют, и метод `crosses()`, который возвращает `True`, если две ломаные пересекаются. Другие методы позволяют вычислять пересечение двух многоугольников, выполнять расширение или сужение геометрий, их упрощать, вычислять расстояние между двумя геометриями и создавать многоугольник, который охватывает все точки в пределах заданного списка геометрий, так называемую *выпуклую оболочку* (convex hull).

Отметим, что `Shapely` – это библиотека управления *пространственными*, а не геопространственными данными. В ней не предусмотрено понятие географических координат. Вместо этого в ней предполагается, что геоданные, прежде чем ими начнут управлять, уже спроецированы, то есть перенесены, на двумерную координатную плоскость, и результаты при желании могут быть конвертированы в географические координаты.

Пример использования

Следующая ниже программа создает два геометрических объекта `Shapely` – круг и квадрат – и вычисляет их пересечение. Пересечением является многоугольник в форме четверти круга, как обозначено темно-серым участком на следующей ниже диаграмме:



Ниже приведен исходный код программы:

```
import shapely.geometry
import shapely.wkt

pt = shapely.geometry.Point(0, 0)
circle = pt.buffer(1.0)

square = shapely.geometry.Polygon([(0, 0), (1, 0),
                                   (1, 1), (0, 1),
                                   (0, 0)])

intersect = circle.intersection(square)
for x,y in intersect.exterior.coords:
    print(x,y)
print(shapely.wkt.dumps(intersect))
```

Исходный код этой программы можно скачать. Ищите файл с именем `shapely_example.py` в пакете примеров программного кода, прилагаемом к данной главе.

Отметим способ создания круга – берется геометрия точки и используется метод `buffer()`, который создает многоугольник, представляющий контур круга.

Документация

Подробная и всесторонняя документация по библиотеке `Shapely` находится на странице http://toblerity.org/shapely/manual.html. Определенно стоит почитать это руководство, и если вы не уверены в том, как ваша задача реализуется в `Shapely`, то эта страница должна стать вашим главным справочным источником.

Визуализация геоданных

Очень сложно, если вообще возможно, интерпретировать геоданные, если не придать им визуальную форму, то есть если их не визуализировать. Для преобразования геоданных в изображения требуется подходящий инструмент. Несмотря на то что имеется целый ряд таких инструментов, мы, в частности, выделим библиотеку **Mapnik**.

Библиотека Mapnik

Mapnik – это общедоступная динамическая библиотека для конструирования картографических приложений. Библиотека Mapnik берет геоданные из базы данных PostGIS, файла фигур или из файла любого другого формата, поддерживаемого пакетом `GDAL/OGR`, и превращает его в ясно визуализированные и привлекательные изображения.

Чтобы добиться приличной визуализации изображений, необходимо решить множество сложных вопросов, и библиотека Mapnik хорошо с ними справляется, предоставляя разработчику геоприложений возможность управлять процессом картографирования. Процедурой вывода отдельных геообъектов на карту управляют правила, тогда как общим видом этих объектов управляют «символизаторы».

Библиотека Mapnik позволяет разработчикам создавать таблицы стилей XML, которые управляют процессом создания карты. Так же, как с таблицами стилей CSS, таблицы стилей библиотеки Mapnik предоставляют вам полный контроль над внешним видом визуализации геоданных. Как вариант различные стили можно создавать в ручном режиме.

Непосредственно сама библиотека Mapnik написана на C++, а чтобы можно было обращаться к практически всему функционалу библиотеки из Python, для этого языка разработаны соответствующие привязки. Так как эти привязки содержатся в самом ядре исходного кода, а не добавлены сторонним разработчиком, то поддержка Python уже встроена прямо в библиотеку Mapnik. Этот факт делает Python в высшей степени подходящим языком для разработки приложений на основе библиотеки Mapnik.

В частности, динамическая библиотека Mapnik интенсивно используется в проектах OpenStreetMap (http://openstreetmap.org) и EveryBlock (http://everyblock.com). С учетом того, что результатом работы библиотеки является просто изображение, эта библиотека легко интегрируется в качестве компонента веб-приложения, впрочем, изображение можно выводить непосредственно в окне настольного приложения. Библиотека Mapnik работает одинаково хорошо на настольных компьютерах и с веб-платформами.

Отметим, что для этой книги мы будем использовать библиотеку Mapnik версии 2.2. На момент подготовки настоящего текста была выпущена ее версия 3.0, и предварительно собранные установщики пока отсутствовали. Это не позволило воспользоваться более свежей версией библиотеки Mapnik, и поэтому мы продолжили пользоваться библиотекой Mapnik 2. Если же вы работаете с библиотекой Mapnik 3, то ваш программный код должен продолжить выполняться, так как он полностью обратно совместим с предыдущей версией Mapnik 2.

Инсталляция библиотеки

Динамическая библиотека Mapnik работает во всех основных операционных системах, включая Windows, Mac OS X и Linux. Главный веб-сайт библиотеки находится на http://mapnik.org.

К сожалению, в настоящее время отсутствует установщик библиотеки Mapnik для Windows с поддержкой Python 3. Поэтому если вы будете использовать Windows, то для работы с Mapnik необходимо использовать Python 2.7. Надеемся, что установщик для Mapnik с поддержкой Python 3 будет, наконец, создан, а пока если вы используете Windows, то ваш единственный вариант состоит в том, чтобы вернуться к использованию Python 2.

Чтобы установить библиотеку Mapnik на ваш компьютер, следуйте за приведенными ниже инструкциями:

- в Windows нужно использовать Python версии 2.7. Вы можете скачать 32-разрядный установочный пакет со страницы для скачивания веб-сайта библиотеки, и далее следуйте инструкциям по установке, которые там приведены[1];

Отметим, что в Windows можно попробовать собрать Mapnik из исходных текстов и сделать библиотеку совместимой с Python 3. Однако процесс компиляции Mapnik из исходников довольно нетривиален, и это не та работа, за которую стоит приниматься, если вы не совсем осведомлены о необходимых инструментах и не готовы потратить время на ее выполнение.

- в Mac OS X двоичный установщик с поддержкой Python 3.3 имеется на сайте библиотеки. Просто загрузите файл `mapnik-osx-v2.2.0.dmg`, щелкните по нему дважды, чтобы открыть образ диска, и затем запустите установщик. По завершении его работы следуйте приведенным в файле `README.txt` инструкциям по обновлению вашей системы, с тем чтобы установленный вами Python 3 мог найти двоичные файлы библиотеки Mapnik;
- для машин на основе Linux можно попробовать установить библиотеку Mapnik, воспользовавшись диспетчером пакетов Python `pip`:

```
dnf install python3-mapnik
sudo pip3 install mapnik --upgrade
```

Согласно записи в блоге на странице http://mapnik.org/news/python-bindings, совсем недавно библиотека Mapnik была обновлена для работы с менеджером пакетов `pip3`. Если все же он не работает, то для установки библиотеки можно воспользоваться вашим предпочтительным диспетчером пакетов либо даже попытаться скомпилировать библиотеку из исходников.

По завершении установки работоспособность библиотеки можно проверить, открыв оболочку Python и набрав следующее:

```
import mapnik
print(mapnik.mapnik_version())
```

---

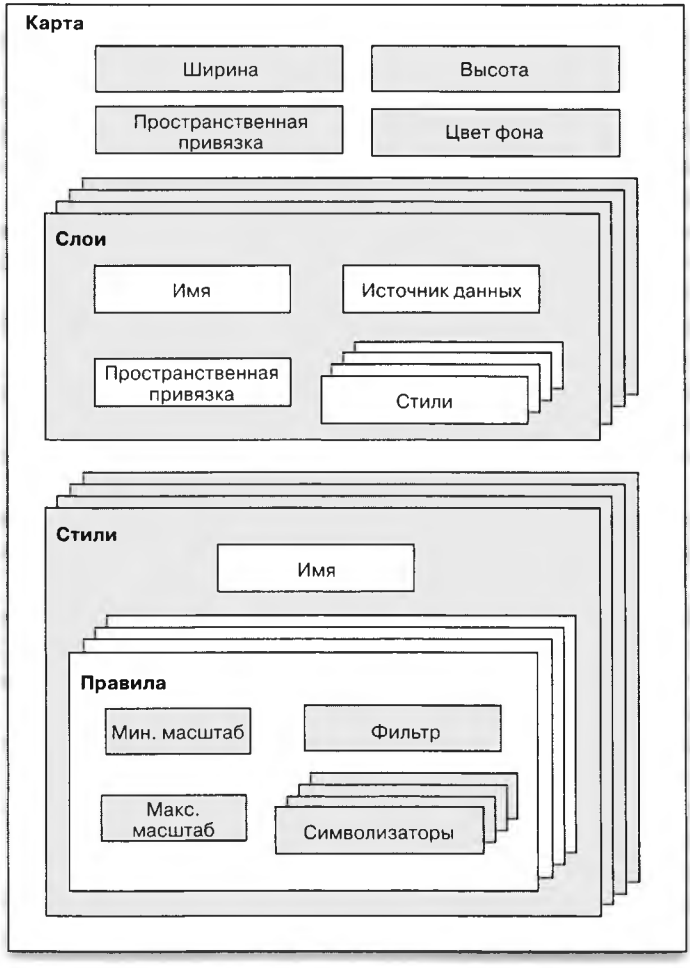
[1] Соответствующие настройки инструментальной среды Spyder для работы с Python 2 подробно рассмотрены в *«Комментарии переводчика»*. – *Прим. перев.*

Результат должен быть целым числом, обозначающим номер версии библиотеки. Например, цифра 200200 соответствует версии 2.2 библиотеки.

Если появляется ошибка, то вам, скорее всего, придется внести изменения в настройки Python, с тем чтобы привязки Python смогли найти базовую динамическую библиотеку Mapnik. Например, возможно, придется отредактировать переменную окружения `pythonpath`, чтобы среда Python смогла обнаружить библиотеку Mapnik.

Концепция библиотеки

В работе с библиотекой Mapnik объект `Map`, представляющий карту в целом, будет основным, с которым вы будете иметь дело. На приведенной ниже схеме показаны компоненты объекта `Map`:



Создавая объект `Map`, вы передаете ему:

- общую ширину и высоту карты в пикселах;
- используемую в карте пространственную привязку;
- цвет фона для заполнения заднего плана содержимого карты.

Затем вы определяете один или несколько тематических слоев **Layer**, которые содержат наполнение карты. Каждый слой имеет:

- имя;
- объект источника данных, определяющий, где получить данные для этого слоя. Источник данных может быть ссылкой на базу данных, файлом фигур либо неким другим источником данных, поддерживаемым пакетом `GDAL/OGR`;
- пространственную привязку, которая используется в этом слое. В случае необходимости она может отличаться от пространственной привязки, используемой картой в целом;
- список стилей, которые применяются к этому слою. К каждому стилю обращаются по имени, поскольку стили фактически задаются в другом месте (возможно, в таблице стилей XML).

Наконец, вы задаете один или несколько стилей **Styles**, которые сообщают библиотеке Mapnik метод отрисовки различных слоев. У каждого стиля есть **имя** и список правил **Rules**, которые составляют главную часть определения стиля. Каждое правило имеет:

- значения **минимального** и **максимального масштаба** (так называемый знаменатель масштаба). Правило применяется, только если масштаб карты находится в пределах этого диапазона;
- выражение **фильтрации**. Правило применяется только к тем объектам, которые соответствуют этому выражению фильтрации;
- список **символизаторов** `Symbolizer`. Они задают то, каким образом соответствующие геообъекты будут визуализированы на карте.

В библиотеке Mapnik реализовано много разных типов символизаторов, в том числе:

- **символизатор линий** `LineSymbolizer`: используется для проведения «черты» вдоль линии, линейного кольца либо вокруг внешней границы многоугольника;
- **символизатор линейных шаблонов** `LinePatternSymbolizer`: использует содержимое графического файла (который указывается по имени) для проведения черты вдоль линии, линейного кольца либо вокруг внешней границы многоугольника;
- **символизатор многоугольников** `PolygonSymbolizer`: используется для заполнения внутренней области многоугольника;
- **символизатор многоугольных шаблонов** `PolygonPatternSymbolizer`: использует содержимое графического файла (который указывается по имени) для заполнения внутренней области многоугольника;
- **символизатор точек** `PointSymbolizer`: использует содержимое графического файла (который указывается по имени) для отрисовки символа в точке;

- **символизатор текста** `TextSymbolizer`: наносит текст (надпись) для геообъекта. Наносимый текст берется в одном из атрибутов геообъекта, при этом имеются многочисленные варианты управления внешним видом наносимого текста;
- **символизатор растра** `RasterSymbolizer`: используется для отрисовки растровых данных, взятых из любого источника данных библиотеки `GDAL`;
- **символизатор знаков** `ShieldSymbolizer`[1]: рисует надпись и точку одновременно. Аналогичен тому, как для отрисовки изображения используется `PointSymbolizer` и для отрисовки надписи – `TextSymbolizer` с одним исключением: он позволяет совмещать текст с изображением;
- **символизатор строений** `BuildingSymbolizer`: использует псевдо 3D-эффект для отрисовки многоугольника, придавая ему вид трехмерного сооружения;
- **символизатор маркеров** `MarkersSymbolizer`: рисует синие стрелки-указатели, согласно направлению геометрий линии и многоугольника. Этот функционал экспериментальный и предназначается для использования с целью отрисовки односторонних улиц на карте города.

При создании экземпляра символизатора и добавлении его к стилю (непосредственно в программном коде или с помощью таблицы стилей XML) вы передаете ряд параметров, которые задают то, как символизатор должен работать. К примеру, при помощи символизатора многоугольников можно определять цвет заполнения, прозрачность и значение гамма, которое помогает выполнять отрисовку примыкающих многоугольников того же самого цвета, не показывая границ. Например:

```
p = mapnik.PolygonSymbolizer()
p.fill = mapnik.Color(127, 127, 0)
p.fill_opacity = 0.8
p.gamma = 0.65
```

Если правило `Rule`, в котором используется символизатор `Symbolizer`, совпадает с одним или несколькими многоугольниками, то эти многоугольники будут отображены с использованием значений заданного цвета, прозрачности и гамма.

Разные правила, естественно, могут иметь разные символизаторы, а также разные значения фильтрации. Например, можно задать правила, которые отображают страны отличающимся цветом в зависимости от их населения.

Пример использования

Следующий ниже образец программы выводит на экран компьютера простую карту мира, используя для этого библиотеку Mapnik:

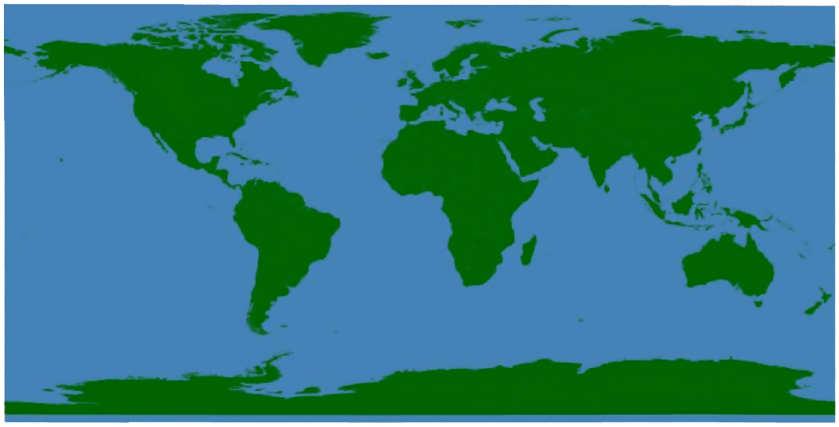
```
import mapnik

symbolizer = mapnik.PolygonSymbolizer()
symbolizer.fill = mapnik.Color("darkgreen"))

rule = mapnik.Rule()
```

[1] Название элемента оформления карт щит (англ. shild) происходит от дорожных знаков в виде маршрутного щита или указателя высокоскоростного шоссе, такого как знак Route 66 в системе нумерованных автомагистралей США. – *Прим. перев.*

```
rule.symbols.append(symbolizer)

style = mapnik.Style()
style.rules.append(rule)

layer = mapnik.Layer("mapLayer")
layer.datasource = mapnik.Shapefile(
                            file="TM_WORLD_BORDERS-0.3.shp")
layer.styles.append("mapStyle")

map = mapnik.Map(800, 400)
map.background = mapnik.Color("steelblue")
map.append_style("mapStyle", style)
map.layers.append(layer)

map.zoom_all()
mapnik.render_to_file(map, "map.png", "png")
```

В этом примере используется набор данных границ стран мира World Borders Dataset, который можно скачать со страницы http://thematicmapping.org/downloads/world_borders.php или по прямой ссылке http://thematicmapping.org/downloads/TM_WORLD_BORDERS-0.3.zip.

Отметим, что эта программа[1] создает символизатор многоугольников `PolygonSymbolizer` для визуализации многоугольников стран и затем прикрепляет символизатор к правилам (объект `Rules`), которые потом становятся составной частью стиля (объект `Style`). Далее мы создаем слой (объект `Layer`), считывая относящиеся к нему данные географической карты из источника данных, находящегося в файле фигур. И в самом конце создаем объект `Map`, к которому присоединяем слой, и полученную карту записываем в графический файл формата PNG:



Исходный код этой программы находится в файле `mapnik_example.py`.

[1] Этот сценарий выполняется только в среде Python 2 (в данном случае 2.7.12). См. настройки среды Spyder в *«Комментарии переводчика»*. – *Прим. перев.*

Документация

К сожалению, веб-сайт библиотеки Mapnik усложняет задачу поиска онлайн-документации. Посвященная библиотеке Mapnik вики-книга находится на странице https://github.com/mapnik/mapnik/wiki. Она представляет собой превосходный источник полезной информации.

Библиотеке Mapnik посвящен ряд документации, расположенной на http://mapnik.org/docs, которая специально предназначена для разработчиков на Python. К сожалению, она довольно скудная и запутанная. Дело в том, что Python-ориентированная документация вытекает из документации для программистов на C++ и сфокусирована на описании того, каким образом реализованы привязки Python, а не на том, каким образом конечный пользователь будет работать с библиотекой, используя этот язык. Вытекающее из этого обилие технических подробностей, которые не относятся к программированию на Python, и отсутствие необходимого объема специфических для Python описаний делают ее не особо полезной.

Начинать работу с библиотекой Mapnik лучше всего, следуя инструкциям по ее установке и затем обратившись к двум предоставленным учебным руководствам. Затем можно просмотреть страницу Learning Mapnik («Изучаем библиотеку Mapnik») в вики-книге по библиотеке Mapnik (https://github.com/mapnik/mapnik/wiki/LearningMapnik) – примечания на этих страницах довольно краткие и непонятные, но там имеется полезная информация, в случае если вы готовы потратить время, чтобы ее найти.

Также стоит обратить внимание на документацию по API Python, несмотря на ее ограниченность. На главной странице перечислены различные классы и ряд полезных функций, многие из которых задокументированы. В самих классах перечислены доступные методы и атрибуты, и даже при том, что при описании многих из них ощущается нехватка в Python-ориентированной документации, как правило, в их функционале можно разобраться из контекста.

Глава 7 «Генерироваине карт при помощи Python и библиотеки Mapnik» содержит подробное описание библиотеки Mapnik; возможно, она окажется более полезной, чем любая другая документация на основе Python.

Заключение

В этой главе мы рассмотрели многие важные библиотеки для разработки геоприложений на основе Python. Мы исследовали динамические библиотеки `GDAL` и `OGR`, которые позволяют читать и писать геоданные с использованием множества форматов. Мы также рассмотрели, как пользоваться библиотекой Python `pyproj` для работы с картографическими проекциями и датумами и как библиотека Python `Shapely` позволяет легко представлять и обрабатывать данные геометрий. Затем мы рассмотрели, как при помощи библиотеки Mapnik можно генерировать привлекательные карты.

Хотя эти инструменты чрезвычайно мощные, вы ничего с ними не добьетесь, если у вас отсутствуют необходимые для работы геоданные. Если вы не настолько удачливы, чтобы иметь доступ к своему собственному источнику данных, либо не горите желанием платить большие суммы для приобретения данных на коммерческой основе, то ваш единственный вариант – использовать общедоступные геопространственные данные из Интернета. Именно такие свободные источники геоданных составляют тему следующей главы.

Глава 4

Источники геоданных

Во время создания геоприложения данные, которые вы используете, важны в той же степени, что и программный код, который вы пишете. Высококачественные геоданные, и в особенности базовые карты[1] и снимки, будут краеугольным камнем вашего приложения. Если ваши карты не будут смотреться хорошо, то ваше приложение будет восприниматься как любительское, независимо от качества написанной вами остальной части программы.

Традиционно геоданные рассматривались как ценный и дефицитный ресурс, реализуемый на коммерческой основе за многие тысячи долларов и со строгими лицензионными ограничениями. К счастью, благодаря тенденции в сторону «демократизации» геопространственных инструментов геоданные теперь все более и более становятся доступными бесплатно и с минимальным ограничением на их использование. Пока еще остаются ситуации, где вам, вероятно, придется за данные заплатить, например чтобы обеспечить качество данных или если вам нужно что-то, что недоступно в другом месте. В целом, однако, данные, в которых вы нуждаетесь, можно легко и совершенно бесплатно скачать с подходящего сервера.

В этой главе предлагается обзор нескольких главных общедоступных источников геоданных такого рода. Приведенный перечень источников ни в коем случае не является исчерпывающим. Скорее, он предоставляет информацию о тех источниках, которые, вероятно, будут самыми полезными для разработчика геоприложений на Python.

В этой главе мы затронем следующие темы:

- несколько главных свободных источников векторных геоданных;
- несколько главных свободных источников растровых геоданных;
- свободные источники геоданных других типов с акцентом на базах данных городов и других топонимов.

---

[1] Подложка (базовая карта, географическая основа карты, англ. термин basemap) – растровая карта, которая применяется в ГИС в качестве фона для наложения и показа растровых и векторных слоев изображения. Используется для привязки данных, нанесения тематического содержания, ориентирования при работе с картой. Существуют три главных типа подложек – ортотрансформированные аэрофото- и спутниковые снимки и отсканированные карты. – *Прим. перев.*

Источники геоданных в векторном формате

Векторно-ориентированные геоданные описывают физические объекты в виде коллекций точек, линий и многоугольников. Нередко с этими геообъектами связаны метаданные.

В этом разделе мы рассмотрим несколько главных источников бесплатных геоданных в векторном формате.

Геоданные проекта OpenStreetMap

Проект OpenStreetMap (http://openstreetmap.org), дословно «открытая карта улиц», – это веб-сайт, где пользователи могут сотрудничать в целях создания и редактирования геоданных. Он характеризуется как «бесплатная, доступная для редактирования карта мира, которая создается волонтерами, в основном любителями, и выпускается с лицензией открытого контента»[1].

Следующий снимок экрана показывает часть карты поселка Онкан, расположенного на острове Мэн, на основе данных проекта OpenStreetMap:

[1] Открытый контент (open content) – любое творческое произведение или контент, опубликованный под лицензией, которая явно разрешает копирование и изменение этой информации кем угодно, а не только закрытой организацией, фирмой или частным лицом. Является альтернативной парадигмой использованию копирайта для создания монополий, способствующих целям демократизации знаний. – *По материалам Википедии.*

Формат данных

Для хранения своих данных проект OpenStreetMap не использует стандартный формат, такой как файлы фигур. Вместо этого там разработан свой собственный на основе XML, который описывает геоданные в виде одиночных точек **node**, линий **way**, то есть последовательностей точек, которые задают линию, многоугольников **area**, то есть замкнутых линий, при помощи которых изображаются многоугольники, и связей **relation**, то есть коллекций других элементов. Любой элемент (точка, линия или связь) может иметь ряд ассоциированных с ним тегов **tag**, которые предоставляют дополнительную информацию об этом элементе в виде пар ключ-значение.

Ниже приведен пример того, как выглядят данные XML проекта OpenStreetMap:

```
<osm>
  <node id="603279517" lat="-38.1456457"
    lon="176.2441646".../>
  <node id="603279518" lat="-38.1456583"
    lon="176.2406726".../>
  <node id="603279519" lat="-38.1456540"
    lon="176.2380553".../>
  ...
  <way id="47390936"...>
    <nd ref="603279517"/>
    <nd ref="603279518"/>
    <nd ref="603279519"/>
    <tag k="highway" v="residential"/>
    <tag k="name" v="York Street"/>
  </way>
  ...
  <relation id="126207"...>
    <member type="way" ref="22930719" role=""/>
    <member type="way" ref="23963573" role=""/>
    <member type="way" ref="28562757" role=""/>
    <member type="way" ref="23963609" role=""/>
    <member type="way" ref="47475844" role=""/>
    <tag k="name" v="State Highway 30A"/>
    <tag k="ref" v="30A"/>
    <tag k="route" v="road"/>
    <tag k="type" v="route"/>
  </relation>
</osm>
```

Получение и использование данных

Геоданные из проекта OpenStreetMap можно получить одним из трех способов:

- чтобы загрузить подмножество данных, которыми вы интересуетесь, можно воспользоваться программным интерфейсом OpenStreetMap (API);
- можно загрузить всю базу данных OpenStreetMap, именуемую `Planet.osm`, и работать с ней локально. Отметим, что объем данных для скачивания измеряется несколькими десятками гигабайт;

- можно воспользоваться одним из зеркал веб-сайта, которые предоставляют данные проекта OpenStreetMap, удобно упакованные в блоки меньшего объема и преобразованные в другие форматы данных. Например, можно скачать данные, относящиеся к Северной Америке, по каждому отдельному штату и в различных форматах, включая файлы фигур.

Приглядимся поближе к каждому из этих трех приемов.

Прикладной программный интерфейс OpenStreetMap

Проект OpenStreetMap предлагает к использованию два API:

- при помощи **редакционного API** (http://wiki.openstreetmap.org/wiki/API_v0.6) можно получать наборы данных OpenStreetMap и вносить в них изменения;
- **просмотровый API Overpass** (http://wiki.openstreetmap.org/wiki/Overpass_API) предоставляет интерфейс только для чтения данных OpenStreetMap.

Просмотровый API Overpass чаще всего используется для получения данных. Он предоставляет тщательно продуманный язык запросов, которым можно воспользоваться для получения данных XML проекта OpenStreetMap на основе критериев поиска, которые вы определяете сами. Например, при помощи следующего ниже запроса можно получить список основных дорог в пределах заданной ограничительной рамки:

```
way[highway=primary](-38.18, 176.13, -38.03, 176.38);out;
```

При помощи библиотеки `python-overpy` (https://github.com/DinoTools/python-overpy) можно вызывать просмотровый API Overpass непосредственно из программ на Python.

База данных Planet.osm

Если вы хотите скачать весь проект OpenStreetMap для работы на своем компьютере, то нужно скачать всю базу данных `Planet.osm`. Эта база данных имеется в двух форматах: в формате сжатого XML-файла, содержащая все точки, линии и связи из базы данных OpenStreetMap, либо в специальном двоичном формате PBF, который содержит ту же самую информацию, но меньшего объема и с меньшим временем чтения.

База данных `Planet.osm` в настоящее время имеет объем 44 Гбайт, если скачивать в формате сжатого XML, либо 29 Гбайт в формате PBF. Файлы обоих форматов можно скачать с http://planet.openstreetmap.org.

Весь дамп базы данных `Planet.osm` обновляется еженедельно, при этом отдельные файлы генерируются еженедельно, ежедневно, ежечасно и ежеминутно. Их можно использовать для обновления имеющейся копии базы данных `Planet.osm` без необходимости скачивать все заново.

Зеркала сайта и выдержки из базы

Из-за объема скачиваемых данных авторы проекта OpenStreetMap рекомендуют для скачивания базы данных `Planet.osm` вместо собственных серверов ис-

пользовать зеркальные веб-сайты. Кроме того, из базы данных предоставляются выдержки, которые позволяют скачивать только те данные, которые относятся к заданной территории, а не ко всему миру. Зеркала и выдержки из базы данных поддерживаются сторонними лицами; за списком их URL-адресов обращайтесь на http://wiki.openstreetmap.org/wiki/Planet.osm.

Отметим, что эти зеркала и выдержки нередко предлагаются в альтернативных форматах, включая файлы фигур и прямые дампы базы данных.

Работа с данными OpenStreetMap

После скачивания базы данных `Planet.osm` на вашем жестком диске разместится огромный файл – в настоящее время это 576 Гбайт, в случае если загружать данные в формате несжатого XML. Имеются два основных варианта для обработки этого файла при помощи Python:

- для чтения файла и извлечения необходимой информации вы можете воспользоваться библиотекой Python, такой как `imposm.parser` (http://imposm.org/docs/imposm.parser/latest);
- вы можете импортировать данные в СУБД и затем обратиться к ним из Python.

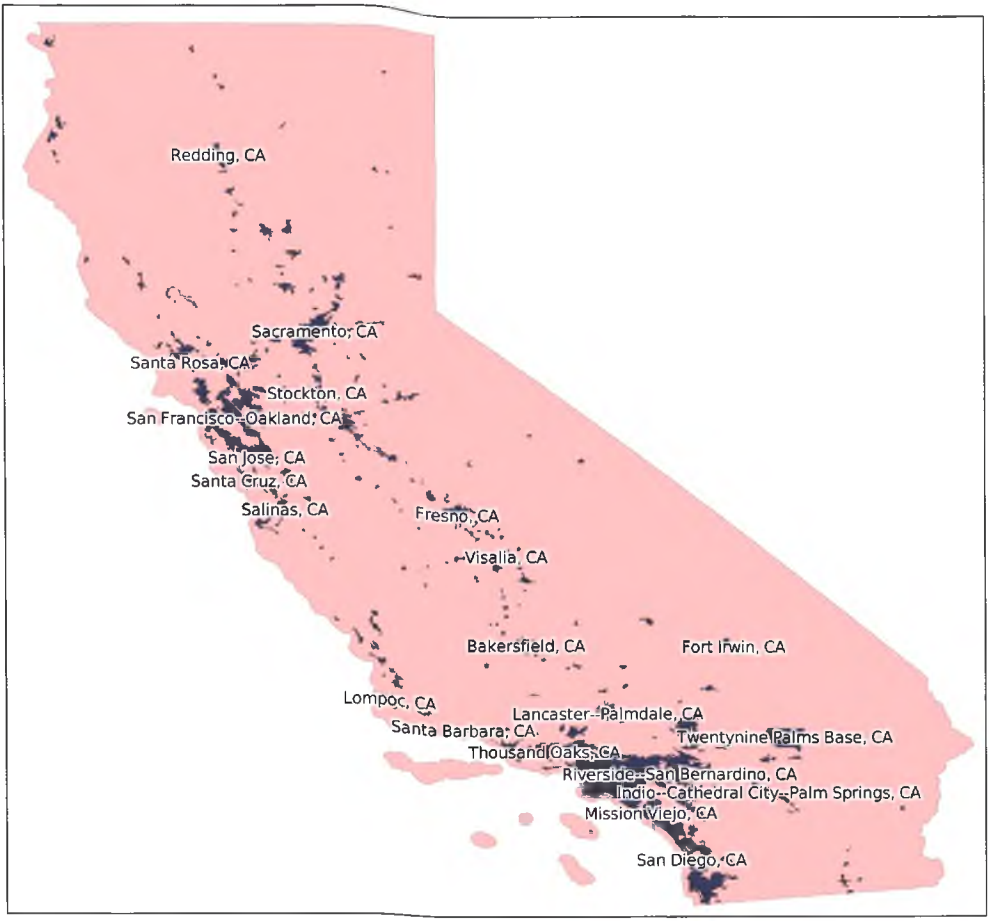
В большинстве случаев, перед тем как вы попробуете работать с данными, вы захотите импортировать их в СУБД. Для этого используйте замечательный инструмент `osm2pgsql`, который имеется на странице http://wiki.openstreetmap.org/wiki/Osm2pgsql.osm2pgsqlwas, созданный, чтобы импортировать все данные `Planet.osm` в СУБД PostgreSQL, и поэтому высоко оптимизированный для этой задачи.

Импортировав данные `Planet.osm` в СУБД на своем компьютере, можно воспользоваться адаптером СУБД `psycopg2` для Python, как описано в *главе 6 «Пространственные базы данных»*, который обеспечивает доступ к данным проекта OpenStreetMap из программ на Python.

База данных TIGER

Бюро переписи населения США сделало доступным большой объем геоданных так называемой **системы топологически интегрированной географической кодировки и привязки** под аббревиатурой **TIGER** (Topologically Integrated Geographic Encoding and Referencing System). База данных TIGER описывает географическую структуру США и включает в себя информацию об улицах, железных дорогах, реках, озерах, географических границах, административных территориях и статистических ареалах, включая школьные округа и городские агломерации. Кроме того, отдельно для скачивания доступны файлы с картографическими данными о границах и демографии.

Следующий ниже рисунок иллюстрирует контур шт. Калифорния и контуры его городских агломераций на основании данных, загруженных с веб-сайта TIGER:



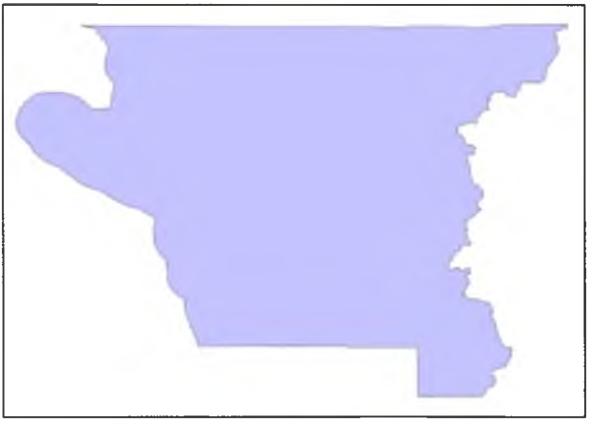
Поскольку данные TIGER производятся американским правительством, они содержат информацию, которая касается исключительно США и их протекторатов (Пуэрто-Рико, Американское Самоа, Северные Марианские острова, Гуам и американские Виргинские острова). База данных TIGER является превосходным источником геоданных для этих территорий.

Формат данных

Вплоть до 2006 г. Бюро переписи населения США предоставляло данные TIGER в собственном текстовом формате под названием **TIGER/Line**. Файлы TIGER/Line хранят каждый тип записи в отдельном файле и для обработки требуют специальных инструментов. К счастью, в случае если вам придется их считывать, библиотека OGR поддерживает файлы в этом формате.

С 2007 г. все данные TIGER производятся в виде файлов фигур, которые (несколько странно) названы файлами фигур TIGER/Line.

Вы можете скачать обновленные файлы фигур с такими геоданными, как диапазоны уличных адресов, ориентиры, переписные участки, метрополисные статистические ареалы и школьные округа. Например, файл фигур со *статистическими ареалами вокруг городского центра*, англ. термин Core-Based Statistical Areas (CBSA), содержит контуры всех метрополисных и микрополисных ареалов, или городских агломераций, США:

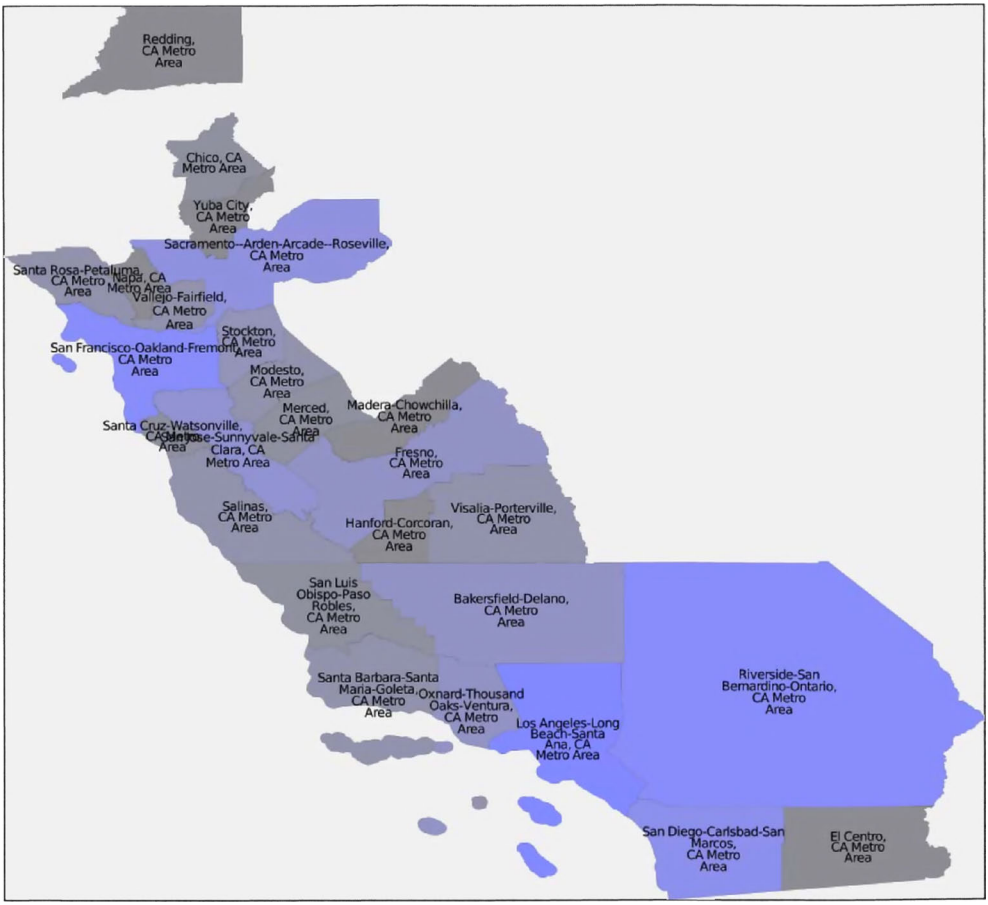


С этим конкретным геообъектом связаны следующие метаданные:

Название атрибута	Значение
ALAND	2606282680.0
AWATER	578761025.0
CBSAFP	18860
CSAFP	None
GEOID	18860
INTPTLAT	+41.7499033
INTPTLON	-123.9809983
LSAD	M2
MEMI	2
MTFCC	G3110
NAME	Crescent City, CA
NAMELSAD	Crescent City, CA Micro Area

Информация по этим разнородным атрибутам находится в обширной документации на веб-сайте TIGER.

Можно также скачать файлы фигур, которые содержат такие демографические данные, как население, число домов, средний возраст и расовый состав жителей. Например, следующая ниже карта окрашивает каждую метрополисную городскую агломерацию в шт. Калифорния в соответствии с общей численностью ее населения согласно данным последней (2010 года) переписи:



Получение и использование данных

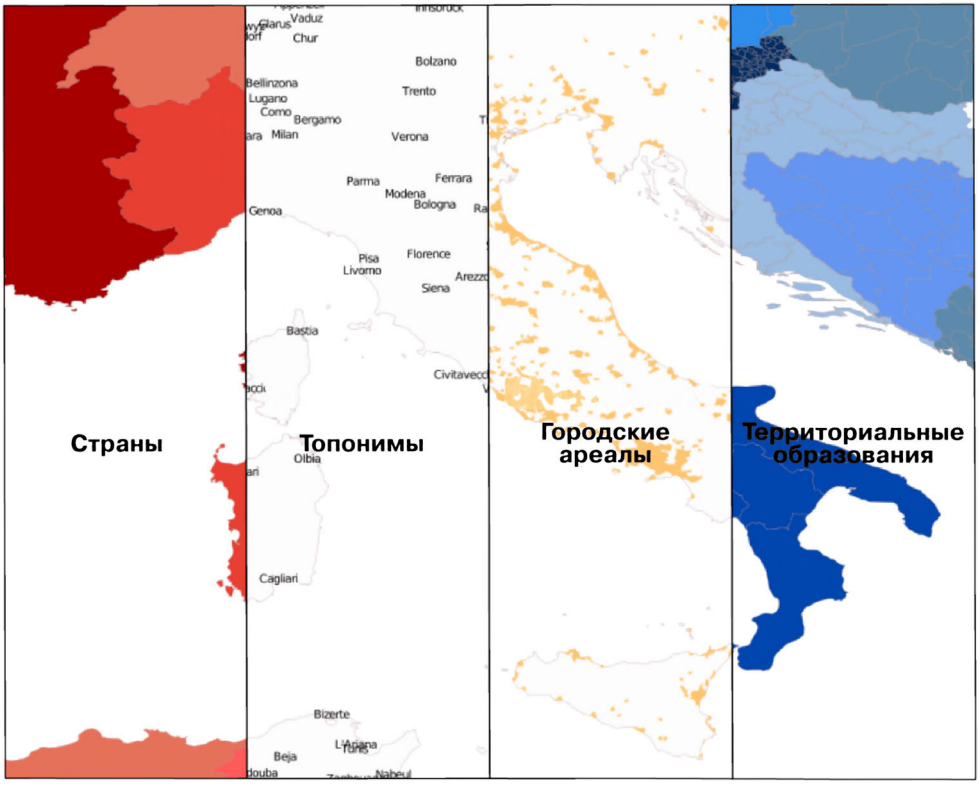
Файлы данных TIGER можно скачать с http://www.census.gov/geo/maps-data/data/tiger.html.

Убедитесь, что вы скачали техническую документацию, так как в ней описаны различные доступные для скачивания файлы и все атрибуты, связанные с каждым объектом. Например, если вы хотите скачать текущий набор городских агломераций США, то файл фигур, который вы ищете, находится в архиве ZIP под названием `tl_2015_us_uaci0.zip` и содержит такую информацию, как название города или поселения и размер городской агломерации в квадратных метрах.

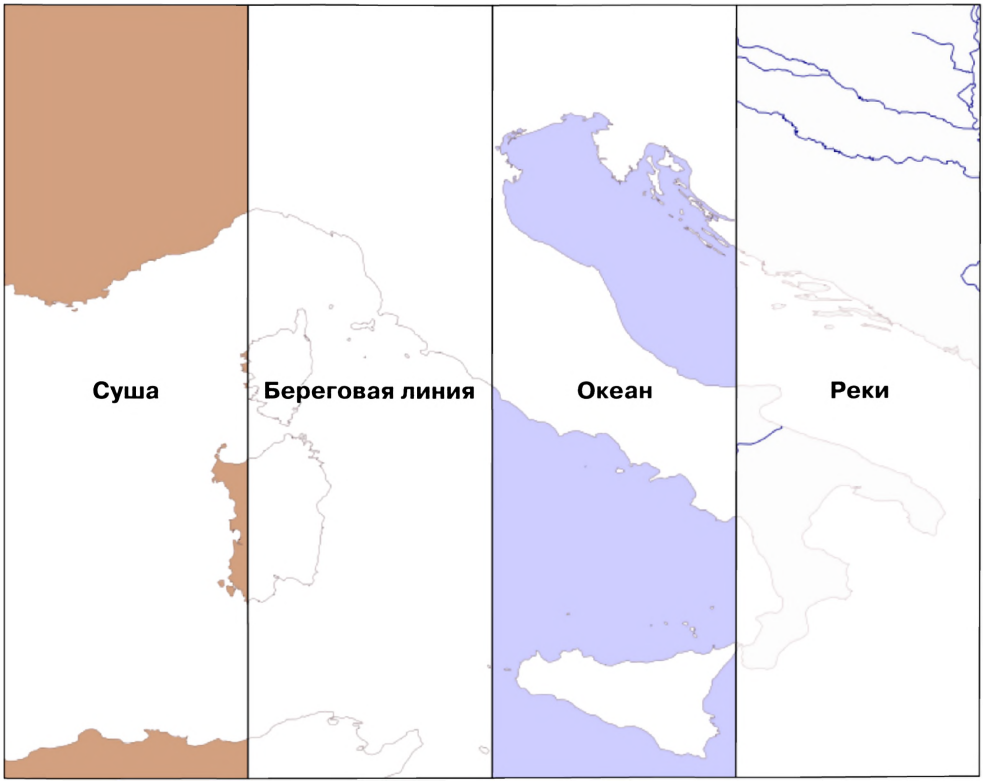
Геоданные веб-сайта Natural Earth

Веб-сайт Natural Earth (http://www.naturalearthdata.com), чье название можно перевести как «Земля, какая она есть», предлагает публично доступные векторные и растровые данные географических карт с высоким, средним и низким разрешениями. Предусмотрены два типа векторных данных географических карт:

- **данные социально-экономических карт**: содержат многоугольники, относящиеся к странам, штатам или регионам, городским агломерациям и контурам национальных парков, а также точечно-линейные данные населенных пунктов и транспортной сети, то есть автомобильных, железных и прочих дорог:



- **данные физических карт**: содержат многоугольники и ломаные континентальных массивов, береговых линий, океанов, небольших островов, рифов, рек, озер и т. д.



Все это можно скачать и свободно использовать в ваших геопространственных программах, что делает веб-сайт Natural Earth превосходным источником данных для вашего приложения.

Формат данных

Все векторные данные веб-сайта Natural Earth предоставляются в формате файлов фигур. Все данные находятся в географических координатах (широты и долготы) с использованием стандартного датума WGS84, что значительно облегчает использование этих файлов в вашем собственном приложении.

Получение и использование векторных данных

Подход создателей веб-сайта к решению этого вопроса в равной степени превосходен, и скачать требуемые файлы очень легко: просто щелкните по ссылке **Get the Data** (Получить данные) на главной странице. Затем вы можете выбрать разрешение и тип нужных данных. Помимо этого, можно также сделать выбор в пользу скачивания отдельного файла фигур или ряда файлов фигур в комплекте. После того как вы их скачали, для работы с содержимым этих файлов фигур можно воспользоваться библиотеками Python, которые обсуждались в предыдущей главе.

Веб-сайт Natural Earth весьма обстоятельный; он содержит подробную информацию о геоданных, которые можно загрузить, и форум, где можно задать вопросы и обсудить любые проблемы, которые у вас есть.

Географическая база данных GSHHG

Американский национальный геофизический центр обработки данных (подразделение Национального управления океанических и атмосферных исследований, NOAA) предоставляет высококачественные векторные данные береговых линий всего мира в виде базы данных, которая называется **Глобальной последовательной иерархической географической базой данных высокого разрешения**, англ. термин Global Self-consistent, Hierarchical, High-resolution Geography Database (GSHHG). Эта база данных содержит подробные векторные данные для береговых линий, озер и рек в пяти различных разрешениях. Данные разделены на 4 уровня: границы океанов, границы озер, границы островов на озерах и границы водоемов на островах, которые расположены на озерах.

Следующее изображение показывает береговые линии, озера и острова Европы, взятые из базы данных GSHHG:

Географическая база данных GSHHG была создана на основе двух публично доступных геопространственных баз данных: база данных **World Data Bank II** (Мировой банк данных II) содержит данные береговых линий, озер и рек, в то время как база данных **World Vector Shorelines** (Глобальные векторные береговые ли-

нии) предлагает данные только береговых линий. Поскольку последняя база данных имеет более точные данные, но испытывает недостаток в информации о реках и озерах, эти две базы данных были объединены, чтобы предоставить максимально достоверную информацию. После слияния баз данных авторы отредактировали данные вручную, чтобы сделать ее непротиворечивой и удалить ряд ошибок. В результате получилась высококачественная глобальная база данных сухопутных и водных границ.

Дополнительная информация с описанием процесса, при помощи которого создавались базы данных GSHHG, находится на странице http://www.soest.hawaii.edu/pwessel/papers/1996/JGR_96/jgr_96.html.

Формат данных

В частности, географическую базу данных GSHHG можно скачать в формате файла фигур. Если скачать данные в этом формате, то у вас в общей сложности получится 20 отдельных файлов фигур, один для каждой комбинации разрешения и уровня.

- **Разрешение** обозначает величину детализации на карте:

Значение	Разрешение	Содержит
c	Грубое	Геообъекты более 500 кв. км
l	Низкое	Геообъекты более 100 кв. км
i	Промежуточное звено	Геообъекты более 20 кв. км
h	Высокое	Геообъекты более 1 кв. км
f	Полное	Все геообъекты

- **Уровень** указывает на тип границ, которые содержатся в файле фигур:

Значение	Содержит
1	Границы океанов
2	Границы озера
3	Границы острова на озере
4	Границы водоема на острове, который расположен на озере

Имя файла фигур сообщает о разрешении и уровне содержащихся данных. Например, файл фигур для океанических границ в полном разрешении назывался бы `GSHHS_f_L1.shp`.

База данных GSSHG раньше называлась несколько иначе – глобальной последовательной иерархической базой данных *береговых линий* высокого разрешения, англ. термин Global Self-consistent, Hierarchical, High-Resolution Shoreline Database (GSHHS). Отсюда понятно, почему сокращение GSHHS до сих пор сохраняется в нескольких местах, включая эти имена файлов.

Каждый файл фигур состоит из единственного слоя, содержащего различные многоугольные объекты, образующие заданный тип границы.

Получение базы данных

Главный веб-сайт географической базы данных GSHHG находится на странице http://www.ngdc.noaa.gov/mgg/shorelines/gshhs.html.

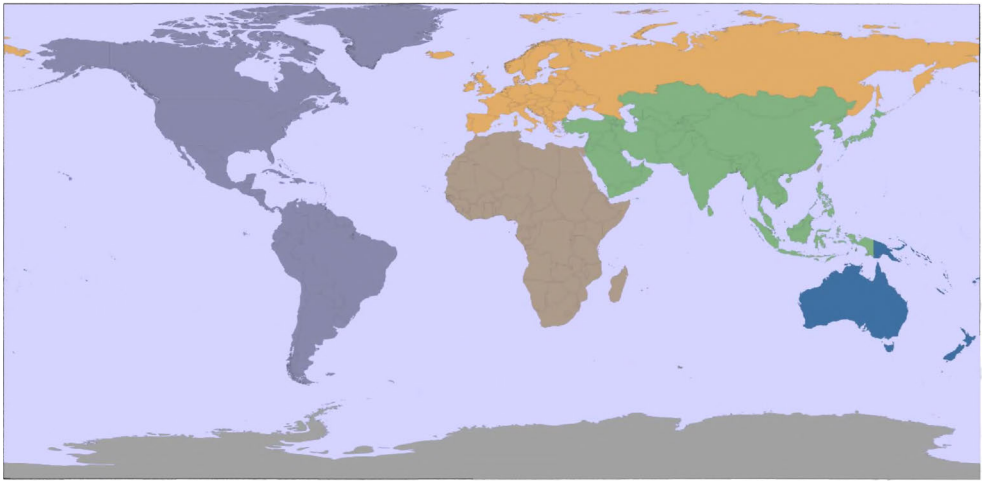
На этой странице перейдите по ссылке для скачивания, чтобы увидеть список файлов, которые можно скачать на свой компьютер. Чтобы скачать данные в формате файла фигур, вам нужен файл `gshhg-shp-2.3.5.zip`, где `2.3.5` – это номер версии скачиваемой базы данных.

После того как вы скачали и распаковали файл, вы можете извлечь данные из отдельных файлов фигур, воспользовавшись для этого библиотекой OGR, как описано в предыдущей главе.

Набор данных границ стран мира

Многие источники данных, которые мы исследовали до сих пор, довольно сложны. Если все, что вы ищете, сводится к неким простым векторным данным, охватывающим весь мир, то набор данных границ стран мира World Borders Dataset может оказаться именно тем, что вам нужно. Хотя границы некоторых стран очевидным образом оспариваются, простота этого набора данных делает его привлекательным кандидатом для большинства основных геоприложений.

Следующая ниже карта была получена на основе набора данных границ стран мира World Borders Dataset:



Набор данных границ стран мира будет активно использоваться на протяжении всей книги. И действительно, вы уже видели пример программы в *главе 3 «Библиотеки Python для геопрограммирования»*, где мы применили библиотеку Mapnik для генерирования карты мира с использованием файла фигур с данными границ стран мира.

Формат набора данных

Набор данных границ стран мира предлагается в виде файла фигур с единственным слоем и одним геообъектом на страну. По каждой стране задан соответствующий геообъект с одним или несколькими многоугольниками, которые очерчивают границу страны, а также полезные атрибуты, включая название страны или территории; несколько идентифицирующих страну кодов, в том числе коды ООН, коды ISO Международной организации по стандартизации и коды FIPS Федеральных стандартов обработки информации, региональную и субрегиональную классификацию, население страны, площадь территории и широту и долготу.

Наличие нескольких кодов облегчает сопоставление объектов с вашими собственными страновыми данными. Кроме того, вы можете использовать такую информацию, как население и территория, для выделения на карте разных стран. Например, на предыдущей карте использовано поле `region`, чтобы выделить каждый географический регион разным цветом.

Получение набора данных

Набор данных границ стран мира World Borders Dataset можно скачать с http://thematicmapping.org/downloads/world_borders.php.

Этот веб-сайт также предоставляет более подробную информацию о содержимом набора данных, включая ссылки на сайт Организации Объединенных Наций, где перечислены региональные и субрегиональные коды.

Источники геоданных в растровом формате

Один из самых захватывающих аспектов таких программ, как Google Планета Земля, состоит в способности «наблюдать» Землю так, как будто вы над ней парите. Это достигается путем изображения тщательно совмещенных между собой аэрофото- и спутниковых снимков, которые создают иллюзию, что вы просматриваете земную поверхность сверху.

Хотя реализовать свою собственную версию приложения Google Планета Земля, скорее всего, у вас не получится, тем не менее имеется возможность получать общедоступные спутниковые снимки в форме растровых геоданных, которые затем можно использовать в собственных геоприложениях.

Правда, только снимками земной поверхности растровые данные не ограничиваются; в растровом формате находится и другая полезная информация – например, **цифровые модели местности** (ЦММ), англ. термин Digital Elevation Models (**DEM**), содержат высоты всех точек, расположенных на поверхности Земли. Их можно использовать для расчета высоты любой нужной точки. При помощи данных ЦММ можно также генерировать двумерные изображения с обозначением различных высот, используя для этого различные оттенки или цвета, либо моде-

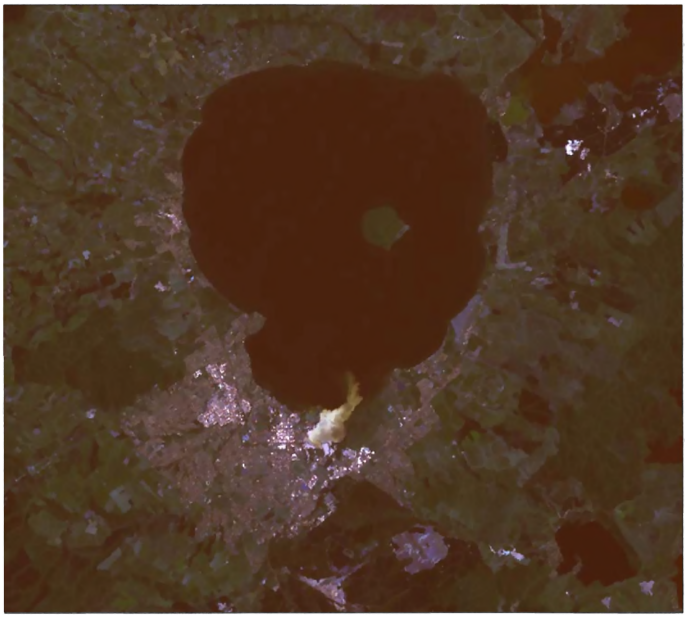
лировать эффект затенения выступов на основе метода так называемого **светотеневого изображения рельефа**[1].

В этом разделе мы рассмотрим самый исчерпывающий источник спутниковой визуальной информации в мире – Landsat, а также растровые данные, которыми располагает веб-сайт Natural Earth, и некоторые общедоступные источники цифровых данных рельефа.

Геоданные проекта Landsat

Landsat – это проект по непрерывной сборке изображения поверхности планеты Земля. Название проекта происходит от слов «земля» (land) и «спутник» (satellite). Группа выделенных спутников на регулярной основе собирает изображения начиная с 1972 г. Визуальная информация проекта Landsat включает в себя черно-белые и традиционные цветные красно-зелено-синие (RGB), а также инфракрасные и термические снимки. Цветные снимки обычно имеют разрешение 30 метров на пиксел, в то время как черно-белые со спутника Landsat 7 – разрешение 15 метров на пиксел.

Ниже показан цветокорректированный спутниковый снимок г. Роторуа, Новая Зеландия. Сам город находится на южном краю озера (внизу фото):



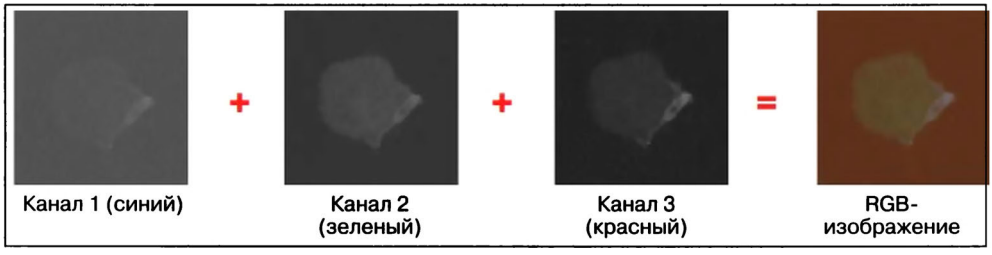
[1] Светотеневое изображение, или пластика, рельефа (shaded relief imagery) – метод изображения меняющейся высоты рельефа при помощи светотени в зависимости от заданного угла и высоты солнца. Также обозначает растровое изображение с таким методом изображения. Синоним *светотеневая отмывка рельефа* восходит к способу изображения рельефа, при котором постепенное изменение силы тени (или цветного тона) достигается отмывкой кистью или тушевкой карандашом. – *Прим. перев.*

Формат данных

Снимки проекта Landsat обычно предлагаются в виде файлов GeoTIFF. GeoTIFF – это файл в графическом формате TIFF с геопространственным тегированием, которое позволяет выполнять привязку изображения к земной поверхности. Большинство программ и инструментов ГИС, включая библиотеку `GDAL`, способно читать файлы в этом формате.

Изображения поступают прямо со спутника, и поэтому файлы, которые вы можете скачать, хранят отдельные каналы данных в отдельных файлах. В зависимости от спутника, с которого поступают данные, может иметься до восьми различных каналов данных – например, спутник Landsat 7 генерирует отдельно красные, зеленые и синие каналы, а также три разных инфракрасных канала, термический канал и «панхроматический» (черно-белый) высокого разрешения.

Чтобы понять, как это работает, приглядимся к процессу создания предыдущего изображения поближе. Исходные спутниковые данные состоят из восьми отдельных файлов GeoTIFF, один для каждого канала. Канал 1 содержит данные в синем цвете, канал 2 – в зеленом, и канал 3 – в красном. Эти отдельные файлы затем можно объединить при помощи библиотеки `GDAL`, которая генерирует единое цветное изображение, как показано ниже:



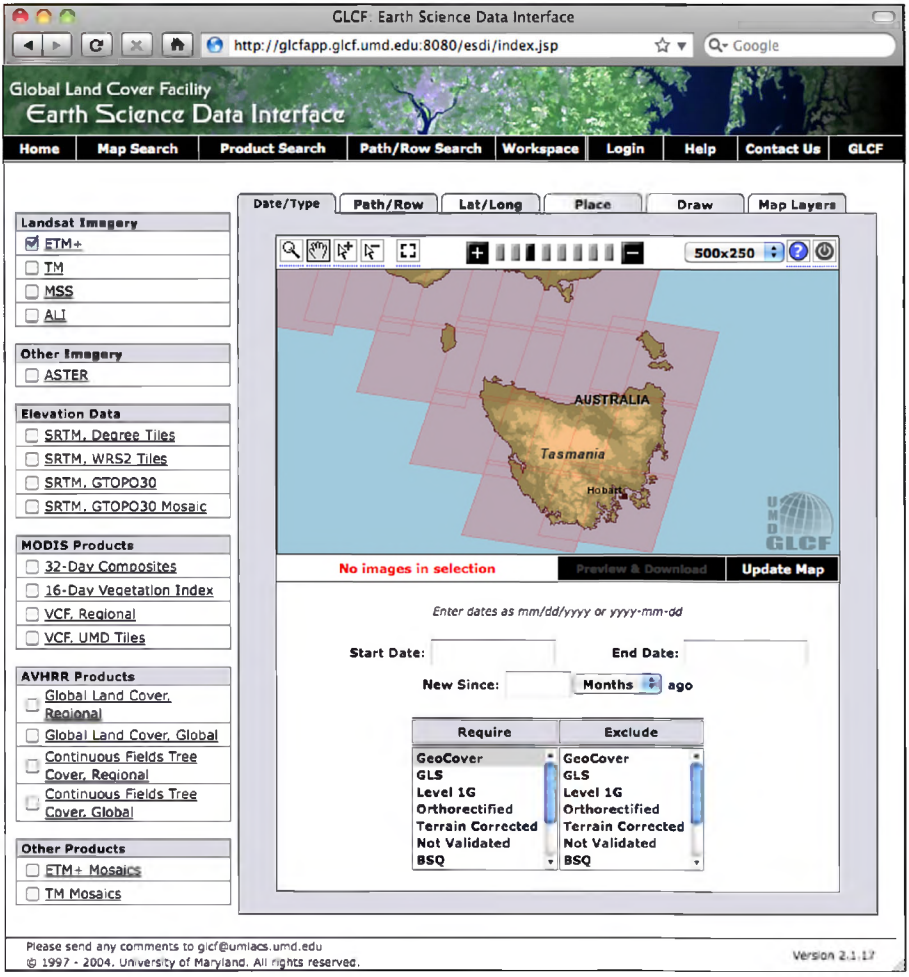
Другая трудность, связанная с данными Landsat, состоит в том, что произведенные спутниками снимки имеют искажения, вызванные различными факторами, включая эллипсоидную фигуру Земли, рельеф фотографируемого ландшафта и ориентацию спутника во время съемки. В связи с этим исходные данные не являются абсолютно точным представлением сфотографированных объектов. К счастью, при помощи процесса, известного как **ортокоррекция изображения**, англ. термин orthorectification[1], можно эти искажения исправить. В большинстве случаев можно сразу скачать ортоскорректированные версии спутниковых снимков.

Получение снимков проекта

Получить доступ к снимкам проекта Landsat легче всего при помощи веб-сайта Центра по изучению глобального почвенно-растительного покрова, англ. название Global Land Cover Facility (GLCF), Университета шт. Мэриленд (http://glcf.umd.

[1] Ортокоррекция снимка, или ортотрансформирование изображения – математически строгое преобразование исходного снимка в ортогональную проекцию и устранение искажений, вызванных рельефом, условиями съемки и типом камеры. – *Прим. перев.*

edu/data/landsat). Щелкните по ссылке **Download via Search and Preview Tool (ESDI)**[1], чтобы перейти в раздел скачивания данных посредством инструмента поиска и предпросмотра, и затем щелкните по кнопке поиска карты **Map Search**. Выберите **ETM+**[2] из списка спутниковых снимков **Landsat Imagery**, и если вы увеличите масштаб нужного участка Земли, то увидите области, покрываемые разными снимками проекта Landsat:

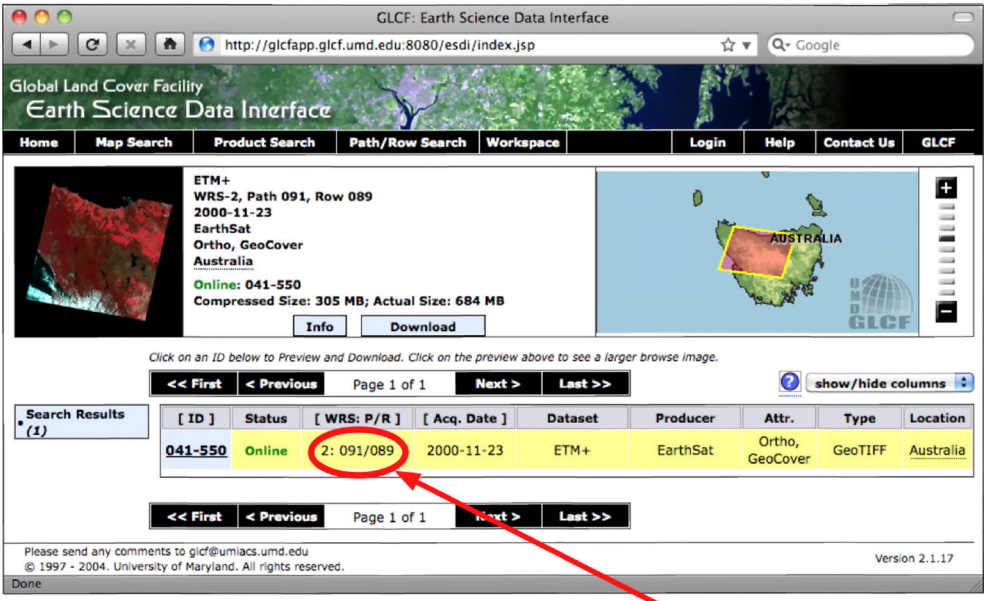


[1] ESDI (Earth Science Data Interface), или интерфейс с данными наук о Земле (с данными геонауки) – это веб-приложение для поиска, просмотра и скачивания данных из онлайновых источников Центра по изучению глобального почвенно-растительного покрова. – *Прим. перев.*

[2] Landsat Enhanced Thematic Mapper Plus (ETM+) – расширенный восьмиканальный мультиспектральный сканирующий радиометр. – *Прим. перев.*

Если выбрать инструмент выделения (), то вы сможете щелкнуть по нужному участку и затем нажать на **Preview & Download** (Предварительный просмотр и скачивание), чтобы выбрать снимок для скачивания.

Как вариант, если вы располагаете данными о номере ряда/колонки (path/row)[1], можно получить доступ к файлам непосредственно через FTP. Номер ряда/колонки, а также номер используемой **глобальной системы индексации WRS**[2] находятся на странице **Preview & Download** (Предпросмотр и скачивание данных):



[1] В глобальной системе индексации данных (WRS), полученных со спутников Landsat, параметр path задает номер «зоны» в направлении с востока на запад, а параметр row определяет номер «зоны» в направлении с севера на юг. Они фактически представляют номер витка (path) спутника по орбите и номер строки (row) относительно широтной осевой линии для нужного участка Земли. Пара значений ряд/колонка представляет многоугольник, соответствующий покрытию сцены Landsat на земле. – *Прим. перев.*

[2] Глобальная система индексации данных (Worldwide Reference System, WRS) – это глобальная схема разграфки, используемая при каталогизации данных, полученных со спутников Landsat. Снимки со спутников Landsat 8 и Landsat 7 используют WRS-2 так же, как это было со спутниками Landsat 5 и Landsat 4. Спутники Landsat 1, Landsat 2 и Landsat 3 используют WRS-1. – *Прим. перев.*

Если вы хотите скачать файлы снимков по FTP, то вот адрес главного ftp-сайта: ftp://ftp.glcf.umd.edu/glcf/Landsat.

Каталоги и файлы имеют сложные имена, состоящие из WRS, номера ряда/колонки, номера спутника, даты съемки и номера канала. Например, файл `p09ir089_7t20001123_z55_nn10.tif.gz` обозначает ряд `091` и колонку `089`, которые, как оказалось, покрывают участок о. Тасмания, выделенный на предыдущем снимке экрана. Число `7` обозначает номер сделавшего снимок спутника Landsat, а `20001123` – это метка даты, обозначающая дату, когда изображение было получено. Заключительная часть имени файла, `nn10`, сообщает о том, что файл предназначен для канала 1.

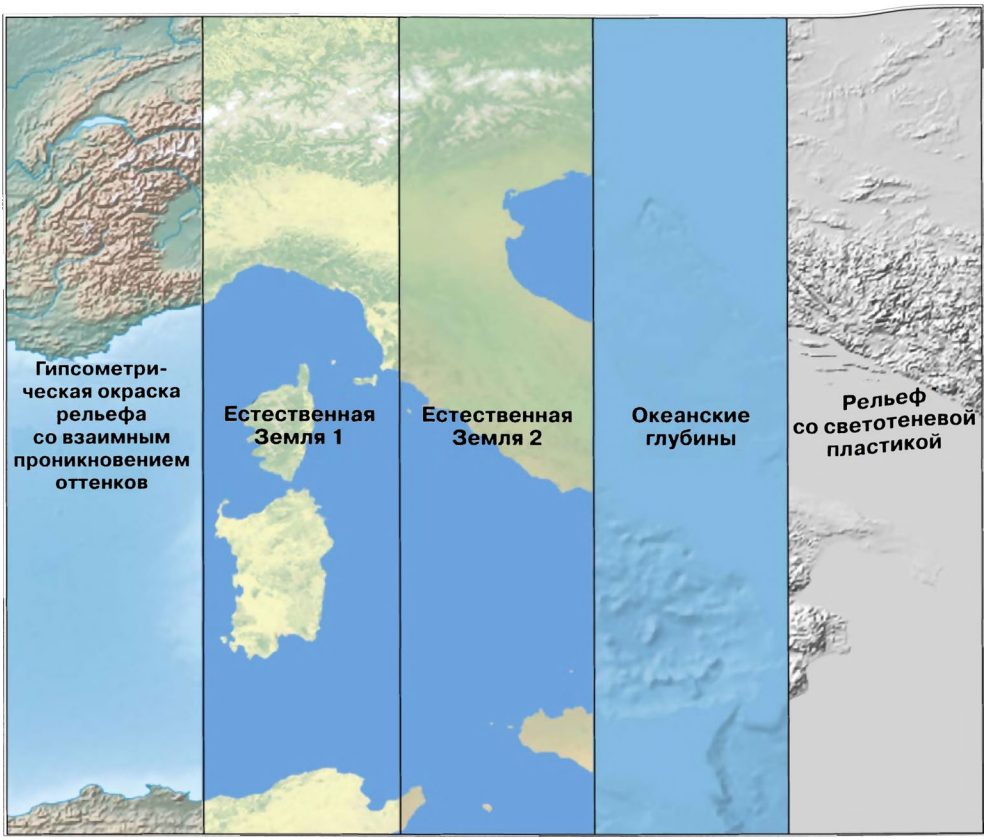
Интерпретируя таким образом имя файла, можно скачивать нужные файлы и подбирать файлы для заданных каналов. Для получения дополнительной информации по поводу того, что означают все эти номера спутников и каналов, обратитесь к ссылкам на документацию в разделе спутниковых снимков **Landsat Imagery** в верхнем правом углу веб-сайта Центра по изучению глобального почвенно-растительного покрова GLCF (http://glcf.umd.edu/data/landsat).

Геоданные веб-сайта Natural Earth

Помимо векторных данных географических карт, веб-сайт Natural Earth (http://www.naturalearthdata.com) предоставляет 5 разных типов растровых карт в масштабах 1:10 млн, 1:50 млн и 1:110 млн.

- **Гипсометрическая окраска рельефа со взаимным проникновением оттенков** обеспечивает визуализацию, при которой цвет выбирается на основе высоты и климата. Полученные изображения зачастую потом комбинируются с изображениями со светотеневой пластикой рельефа для создания реалистичной визуализации поверхности Земли.
- **Natural Earth 1** и **Natural Earth 2** – это более идеализированные визуализации поверхности Земли; в них используются легкая палитра и мягко смешанные цвета, обеспечивающие превосходный фон для отрисовки своих собственных геоданных.
- Набор данных океанских глубин **Ocean Bottom** использует комбинацию светотеневого изображения рельефа с окраской на основе глубины, тем самым обеспечивая визуализацию дна океана.
- Изображения на основе **светотеневой пластики рельефа** используют шкалу полутонов, «оттеняющих» поверхность Земли на основе данных рельефа в высоком разрешении.

Следующий ниже рисунок показывает, как эти растровые карты выглядят:



Кроме того, имеется дополнительный растровый набор данных, который обеспечивает батиметрическую визуализацию (подводных глубин) в масштабе 1:50 млн. Следующая ниже диаграмма – это пример батиметрических данных для океанов, окружающих Новую Зеландию:

Формат данных

Большинство растровых данных веб-сайта Natural Earth находится в стандартном графическом формате TIFF. Исключение составляют батиметрические данные, которые реализованы в виде многослойного файла Adobe Photoshop с различными оттенками синего цвета, связанными с каналами глубины.

Во всех случаях растровые данные находятся в координатной (широта и долгота) проекции, и в них используется стандартный датум WGS84, облегчая трансляцию географических координат широты и долготы в прямоугольные координаты пикселов внутри растрового изображения и обратно.

Получение и использование растровых данных

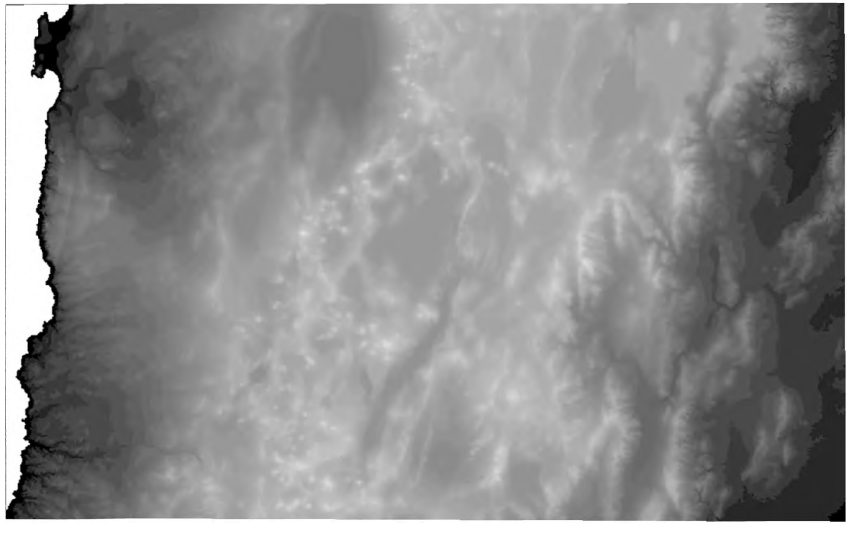
Так же как и векторные данные, растровые данные веб-сайта Natural Earth легко скачать: просто зайдите на веб-сайт (http://www.naturalearthdata.com) и, чтобы скачать растровые данные, следуйте по ссылке **Get the Data** (Получить данные). Можно выбрать масштаб скачиваемых данных 1:10 млн, 1:50 млн либо 1:110 млн, а также их размер: версии каждого файла большого или малого размера.

Скачав файл с данными в формате TIFF, его можно открыть в графическом редакторе или использовать утилиту командной строки, такую как `gdal_translate`, для управления изображением. Что касается данных батиметрии, то вы можете открыть файл прямо в Adobe Photoshop или использовать более дешевую альтернативу, такую как графический редактор GIMP или Acorn от Flying Meat[1]. Каждый канал глубины – это отдельный слой в файле, который по умолчанию связан с определенным оттенком синего цвета. По желанию можно выбрать другие цвета, а также какие слои показывать или скрывать. По завершении работы с изображением его можно свести в один слой и сохранить в формате TIFF для использования в программах.

Геоданные проекта GLOBE

Высококачественная глобальная цифровая модель местности с базой в 1 км на пиксел, англ. название Global Land One-kilometer Base Elevation (GLOBE), – это международный проект по созданию высококачественных глобальных данных **цифровой модели местности** (ЦММ) среднего разрешения. Результатом проекта является ряд общедоступных файлов ЦММ, которые можно использовать для многих типов геоанализа и разработки различных геоприложений.

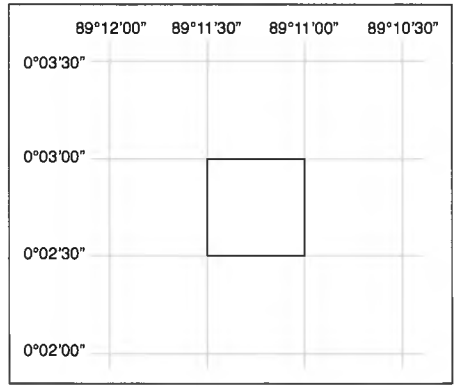
Следующая ниже диаграмма показывает данные ЦММ проекта GLOBE для северной части Чили, преобразованные в полутоновое изображение:



[1] Для быстрого просмотра изображений в Windows лучше всего воспользоваться свободной программой просмотра IfranView (http://www.irfanview.com/). Мультиплатформенный (Linux, OS X, Windows) графический редактор GIMP можно скачать с https://www.gimp.org/. Графический редактор Acorn имеется только для Mac OS X (http://www.flyingmeat.com/acorn/). – *Прим. перев.*

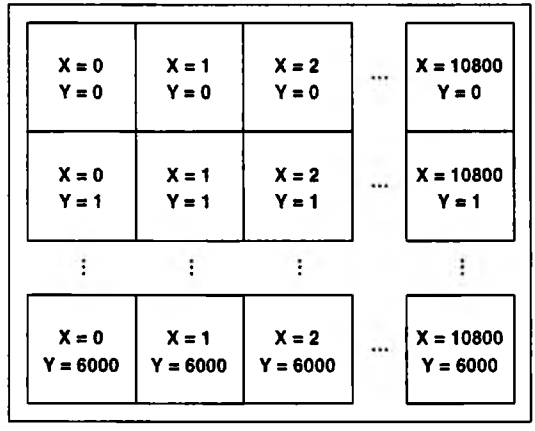
Формат данных проекта

Как и все данные ЦММ, в проекте GLOBE для представления высоты в заданной точке на поверхности Земли используются значения ячеек растровых данных. В случае с GLOBE эти данные состоят из 32-разрядных знаковых целых, представляющих высоту в метрах над (или под) уровнем моря. Каждая ячейка («пиксел») растровых данных содержит высоту квадрата на поверхности Земли, который по долготе составляет 30 угловых секунд (приблизительно 1 км) в ширину и по широте 30 угловых секунд в высоту:



Отметим, что 30 угловых секунд приблизительно равны 0.00833° широты или долготы, что эквивалентно квадрату с шириной и высотой примерно в один километр. Иными словами, данные GLOBE имеют разрешение приблизительно 1 километр на ячейку, или «пиксел».

Данные GLOBE в исходном виде – это просто длинный список 32-разрядных целых чисел в формате с обратным порядком байтов, в котором ячейки читаются слева направо и затем сверху вниз, как показано ниже:



В отдельном заголовочном файле (.hdr) содержится более подробная информация о данных ЦММ, включая ширину, высоту и их географически привязанное местоположение. В случае предоставления заголовочного файла такие инструменты, как библиотека GDAL, способны прочесть исходные данные.

Получение и использование данных

Главный веб-сайт проекта GLOBE находится на странице http://www.ngdc.noaa.gov/mgg/topo/globe.html. Кроме того, там можно скачать подробное руководство, расположенное по адресу http://www.ngdc.noaa.gov/mgg/topo/report/globedocumentationmanual.pdf, описывающее набор данных проекта GLOBE.

Данные модели местности проекта GLOBE предлагаются в виде серии из 16 «сегментов»[1], охватывающих всю поверхность Земли. Перейдя по ссылке **Get Data Online** (Получить данные онлайн) на главной странице, можно загрузить любой из сегментов, который потребуется.

К сожалению, данные сегмента, которые вы скачиваете, содержат только сами исходные данные рельефа. Необходимая для библиотеки GDAL информация о привязке данных рельефа к поверхности Земли и формате, в котором данные хранятся, отсутствует. Если вы не хотите заниматься вычислением этой информации самостоятельно, то нужно скачать файл .hdr, соответствующий каждому сегменту. Файлы .hdr находятся на странице http://www.ngdc.noaa.gov/mgg/topo/elev/esri/hdr.

Скачав данные, просто поместите исходный файл ЦММ в тот же самый каталог, что и файл .hdr. Затем файл можно открыть непосредственно при помощи библиотеки GDAL, как показано ниже:

```
import osgeo.gdal
dataset = osgeo.gdal.Open("j10g.bil")
```

Набор данных будет состоять из единственного канала растровых данных, которые затем можно прочесть при помощи GDAL.

Чтобы увидеть пример использования библиотеки GDAL для обработки данных ЦММ, пожалуйста, обратитесь к разделу GDAL в главе 3 «Библиотеки Python для геопрограммирования».

Национальный набор данных рельефа

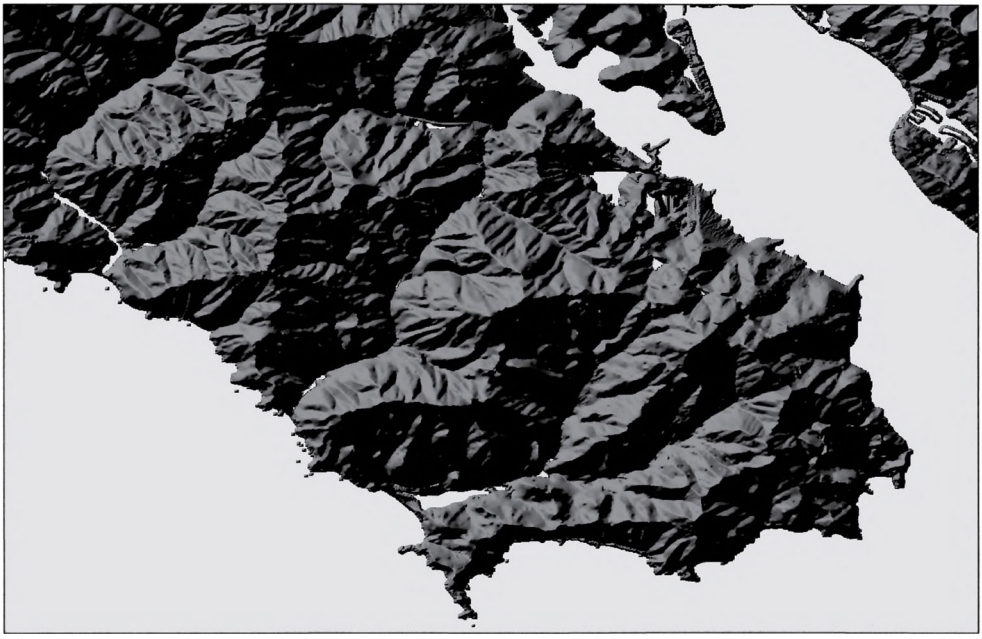
Национальный набор данных рельефа США (НДР), англ. термин National Elevation Dataset (**NED**), – это цифровая модель поверхности Земли с высоким разрешением, предоставленная Геологической службой США. Она охватывает континентальную часть США, Аляску, Гавайи и другие американские территории.

---

[1] Сегмент карты (tile) – это регулярная ячейка (клетка) картографической сетки. Подмножество сегментов образует *лист* (tileset), а все сегменты укладываются в единую *мозаику*, образующую *сборную карту* (tile map). При этом разбивка данных цифровой карты на сегменты называется *нарезкой* (tiling), а соединение сегментов в лист или сборную (мозаичную) карту– склейкой (stiching). – *Прим. перев.*

Значительная часть США покрывается данными рельефа с разрешением около 30 либо 10 метров на ячейку (соответственно, 1 угл. сек. и 1/3 угл. сек.), причем отдельные зоны имеют разрешение 3 метра на ячейку. Данные по Аляске в основном доступны с разрешением 60 метров на ячейку.

Следующее ниже изображение со светотеневой пластикой рельефа было получено при помощи данных рельефа NED для мыса Марин, г. Сан-Франциско:



Формат данных

Данные NED можно скачать в различных форматах, включая IMG, GeoTIFF и ArcGRID, каждый из которых можно обработать при помощи библиотеки `GDAL`.

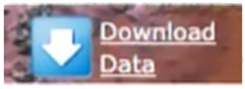
Как и с другими данными ЦММ, каждая ячейка в растровом изображении содержит высоту заданного участка на поверхности Земли. Для данных NED высота исчисляется в метрах выше или ниже опорной высоты, известной как *североамериканский вертикальный датум 1988*, англ. термин North American Vertical Datum (NAVD88). Он примерно эквивалентен высоте над уровнем моря с учетом приливных и других колебаний.

Получение и использование данных

Главный веб-сайт NED находится по адресу http://ned.usgs.gov.

Этот сайт описывает набор данных NED; чтобы скачать данные, необходимо воспользоваться средством просмотра карт **National Map Viewer**, которое предлагается на странице http://viewer.nationalmap.gov/viewer/.

Работая со средством просмотра, увеличьте масштаб интересующего участка и затем щелкните на ссылке **Download Data** (Скачать данные) в верхней части страницы:

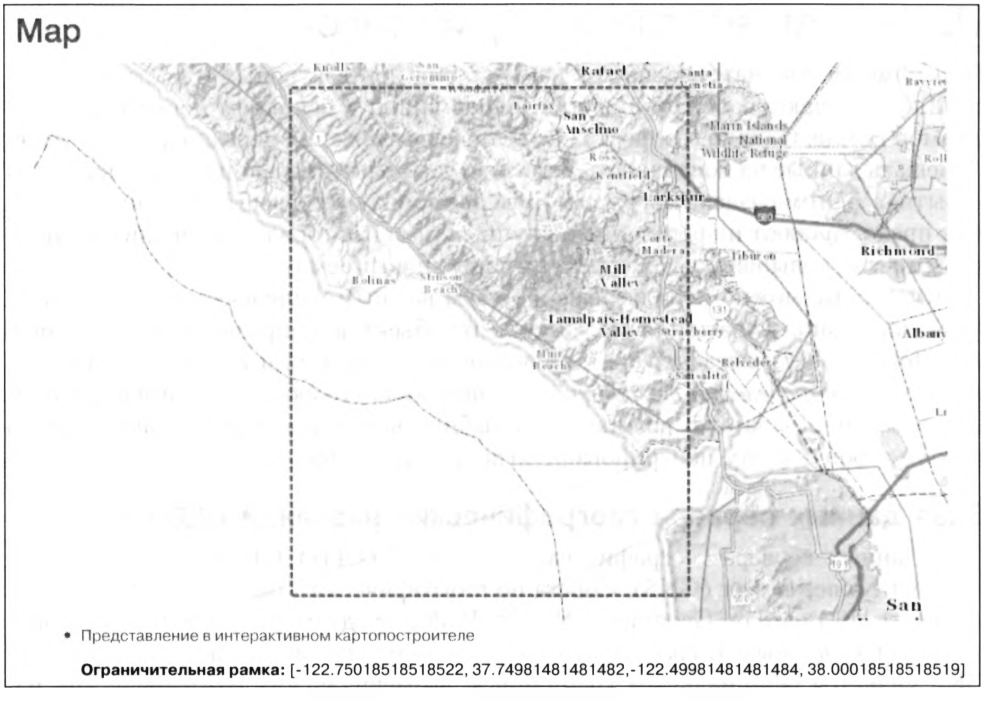


Щелкните по опции, соответствующей скачиванию текущего экстента карты, и затем из предложенного списка выберите опцию **Elevation DEM Products** (Продукты ЦММ с данными рельефа). Нажав кнопку **Next** (Далее), вы получите список доступных наборов данных ЦММ (DEM), которые охватывают видимый участок карты:

Выберите набор данных, которым вы интересуетесь, и щелкните по расположенному рядом значку **Info** (Информация).

Отметим, что при разрешении 1/9 угловой секунды (arcsec) размер ячейки сетки в наборе данных составит приблизительно 3×3 метра.

Когда вы щелкнете по значку **Info**, в вашем браузере откроется новая вкладка с подробной информацией о выбранном наборе данных. В частности, вы увидите карту, которая будет показывать участок, покрываемый этим набором данных:



• Представление в интерактивном картопостроителе

Ограничительная рамка: [-122.75018518518522, 37.74981481481482, -122.49981481481484, 38.00018518518519]

Если прокрутить страницу вниз к разделу ссылок на внешние источники **Links (External Sources)**, то вы увидите прямую ссылку на загрузку набора данных. Щелчок по ней сразу начнет загрузку файла.

Средство просмотра Национальной карты содержит опцию электронной корзины для одновременной загрузки нескольких наборов данных. Тем не менее ее можно проигнорировать и просто загрузить весь набор данных.

В итоге получится сжатый файл в формате `.zip`, содержащий нужные данные, и большое количество файлов метаданных и документации о наборе данных. После распаковки архива `.zip` набор данных можно открыть в библиотеке `GDAL` точно так же, как вы открываете любой другой растровый набор данных:

```
import osgeo.gdal
dataset = osgeo.gdal.Open("dem.tif")
```

Наконец, если вы работаете с данными ЦММ (DEM), вам может понадобиться утилита gdaldem, которая включена в состав пакета GDAL для скачивания. Эта программа облегчает просмотр и управление растровыми данными ЦММ. Предыдущее изображение с теневой пластикой рельефа было создано как раз при помощи этой утилиты, как показано ниже:

```
gdaldem hillshade dem.tif image.tiff
```

Источники геоданных других типов

При помощи рассматривавшихся нами до сих пор векторных и растровых геоданных получают изображения или информацию в основном непосредственно о самой Земле. Однако геопространственные приложения нередко должны уметь помещать данные на поверхность Земли, то есть выполнять привязку к месту или событию. В этом разделе мы рассмотрим две дополнительные базы данных, которые предоставляют информацию о крупных и малых городах, природных объектах и точках повышенного интереса на поверхности Земли.

Эти данные можно использовать по двум важным направлениям. Во-первых, их можно использовать для *маркировки* геообъектов, например поместить метку «Лондон» на географически привязанном изображении южной Англии. Во-вторых, для *определения географического положения* геообъекта по имени, например давая пользователю возможность выбирать город из выпадающего списка и затем строить карту, центрированную вокруг этого города.

База данных сервера географических названий GEOnet

База данных сервера географических названий (СГН) GEOnet, англ. термин GEOnet Names Server (GNS), Совета по географическим названиям США, англ. название US Board on Geographic Names (BGN), предлагает большой набор топонимических данных. В соответствии с решением этой организации база данных GNS является официальным хранилищем неамериканских географических названий.

Ниже приведен фрагмент базы данных сервера географических названий GEOnet:

LAT	LONG	FC	DSG	ELEV	NT	FULL_NAME
-46.333333	168.716667	S	RSTN		N	Kamahl
-46.816667	168.25	T	ISL		N	North Island
-40.959722	175.6575	P	PPL		N	Masterton
-52.556111	169.136667	H	COVE		N	Camp Cove
-39.455556	173.858333	P	PPL		N	Opunake
-52.501339	169.120556	T	ISL	76	N	Gomez Island
-39.591667	174.283333	P	PPL		N	Hawera
-36.641344	175.360 B33	T	RKS		N	Motupotaka Rocks
-41.255833	173.263333	S	TOVYR		N	Boulder Bank Lighthouse

LAT	LONG	FC	DSG	ELEV	NT	FULL_NAME
-36.605276	174.9175	T	RK		N	Shearer Rock
-36.610556	174.705556	P	PPL		N	Red Beach
-46.5625	169.619444	T	ISL		N	Cosgrove Island
-42.133333	171.616667	T	MT		N	Mount McHardy
-39.266667	174.616667	T	MTT		N	Turakiral
-37.433333	174.7	T	HLL		N	Nihonui
-44.033333	169.25	H	STM		N	Cowan Creek
-44.65	170.35	H	STM		N	Deep Creek
-40.966667	173.916667	H	BAY		V	Waltara Bay
-42.183333	173.466667	H	STM		N	Spray Stream
-39.45	173.833333	H	STM		N	Heimama Stream
-46.1	166.466667	H	CHUM		V	Eastern Entrance
-36.766667	174.4	T	BCH		N	Muriwai Beach
-38.983333	177.45	T	HLL		N	Ohlnemaemae
-41.25	174.116667	H	BAY		N	Kahlkatea Bay
-36.583333	174.6	P	PPL		N	Parakakau
-41.152222	173.438333	H	COVE		N	Pier Cove
-39.183333	177.85	H	STM		N	Mangatea Stream
-41.383333	172.583333	T	MT		N	Mount Gomorrah
-42.55	171.966667	H	STM		N	Waiheke River

Как видите, база данных содержит значения долготы и широты, а также коды, обозначающие тип топонима (населенный пункт, административный район, природный объект и т. д.), высоту (где это актуально) и код, обозначающий тип названия (официальное, обычное, историческое и т. д.).

База данных GEOnet содержит приблизительно 6 миллионов геообъектов и 10 миллионов названий, относящихся ко всем странам, кроме США и исключая Антарктиду.

Формат данных

Данные GEOnet предоставляются в виде простого файла, где в качестве разделителя полей использован символ табуляции. Первая строка файла содержит имена полей, а последующие строки – различные геообъекты, один на строку. Импорт этих топонимических данных в электронную таблицу или базу данных тривиален.

Для получения дополнительной информации об имеющихся полях и значении используемых кодов обратитесь к странице http://geonames.nga.mil/gns/html/gis_countryfiles.html.

Получение и использование базы данных

Главный сайт сервера географических названий GEOnet находится на странице http://geonames.nga.mil/gns/html/.

Основное взаимодействие с базой данных GEOnet выполняется посредством различных инструментов поиска, которые позволяют просматривать данные в отфильтрованном виде. Чтобы скачать данные прямиком без поиска, пройдите на страницу http://geonames.nga.mil/gns/html/namefiles.html.

В перечне содержатся все страны; просто щелкните по гиперссылке страны, по которой вы хотите получить данные, и ваш браузер загрузит `.zip` файл, содержащий ряд файлов с разделением полей символом табуляции, содержащий все геообъекты в этой стране. Данные по каждой стране предоставляются одним комбинированным файлом либо отдельными файлами по каждому типу геообъектов.

Помимо скачивания данных индивидуально по каждой стране, также имеется возможность скачать данные по всем странам одним файлом объемом 460 Мб.

После скачивания файлов и их распаковки можно загрузить файлы непосредственно в электронную таблицу или базу данных для последующей обработки. Выполнив фильтрацию по полю классификации объекта **FC** (Feature Classification), кодового обозначения объекта **DSG** (Feature Designation Code) и другим полям, можно отдельно выбрать нужный набор топонимов и затем использовать эти данные непосредственно в своем приложении.

Данные информационной системы географических названий США

Информационная система географических названий (ИСГН), англ. термин Geographic Names Information System (GNIS), – это эквивалент сервера географических названий GEOnet для США. Она содержит топонимическую информацию (географический номенклатурный перечень) исключительно по США.

Ниже приведен фрагмент из базы данных GNIS:

FEATURE_NAME	FEATURE_ CLASS	5TATE_ ALPHA	PRIM_ LAT_DEC	PRIM_ LONG_DEC	ELEVATION
Abbott Ranch	Locale	CA	36.2305176	-121.4657686	250
Abbott Reservo r	Reservoir	CA	40.9060035	-120.8613504	1760
Abbott Spring	Spring	CA	40.9093369	-120.8535725	1794
Abbotts Lagoon	Lake	CA	38.1174233	-122.9533306	0
Abbotts Peak	Summit	CA	37.9763136	-120.6224262	471
Abbotts Upper Cabin	Building	CA	41.4295777	-123.1875457	1477
ABC Camp Rustic Campsite	Locale	CA	36.0232958	-121.4299341	756
ABC-TV Heliport	Airport	CA	34.1033427	-118.2834088	129
Abel Canyon	Valley	CA	34.8233155	-119.8643049	524
Abel Canyon Campground	Locale	CA	34.82276	-119.8626382	524
Abel Canyon Spring	Spring	CA	34.8710918	-119.816803	1190
Abel Square Shopping Center	Locale	CA	37.427717	-121.9080126	5
Abelardo Cabin	Locale	CA	36.3102401	-120.7585092	1146
Abelian Group Math School	School	CA	37.8685219	-122.2876776	23
Abels Apple Acres	Locale	CA	38.7465695	-120.748544	797
Aberdeen	Populated Place	CA	36.9779897	-118.2534321	1193

FEATURE_NAME	FEATURE_ CLASS	5TATE_ ALPHA	PRIM_ LAT_DEC	PRIM_ LONG_DEC	ELEVATION
Aberdeen Bypass Ditch	Canal	CA	36.9616004	-118.2362091	1173
Aberdeen Canyon	Valley	CA	34.1155644	-118.2889647	196
Aberdeen Ditch	Canal	CA	36.9646558	-118.225931	1174
Aberdeen-Inverness Resicence Hall	Building	CA	33.978349	-117.3253209	331
Abernathy Meadow	Flat	CA	37.8752015	-119.8993467	1292
Abestos Number 1 Prospect	Mine	CA	36.8940982	-118.069813	2033
Abilene	Populated Place	CA	36.145507	-119.053714	124
Able Spring	Spring	CA	39.1473909	-122.6310973	924
Abies Drain	Canal	CA	37.4274355	-120.9690959	19
Abney Butte	Summit	CA	41.9759586	-123.1603266	1269
Abolitos Park	Park	CA	32.9833782	-117.0600321	175
Abraham Lincoln Cont nuation High School	School	CA	33.9711265	-117.3661556	275
Abraham Lincoln Elementary School	School	CA	33.6100221	-117.8606119	89

База данных GNIS содержит естественные, физические и культурные (искусственные) объекты, правда, она не включает в себя названия дорог и магистралей.

Аналогично базе данных GEOnet, база данных GNIS в соответствии с решением Совета по географическим названиям США содержит официальные названия, используемые американским правительством. База данных GNIS управляется Геологической службой США и в настоящее время содержит более 2.2 миллиона геообъектов.

Формат данных

Названия топонимов базы данных GNIS доступны для скачивания в виде сжатых текстовых файлов с разделителем в виде символа вертикальной черты (|). В таком формате символ вертикальной черты используется для разграничения полей данных:

```
FEATURE_ID|FEATURE_NAME|FEATURE_CLASS|...
1397658|Ester|Populated Place|...
1397926|Afognak|Populated Place|...
```

Первая строка содержит имена полей, последующие строки – различные геообъекты. Предоставляемая информация включает в себя название геообъекта, его тип, высоту, округ и штат, в котором геообъект расположен, географическую координату широты и долготы самого геообъекта и географическую координату широты и долготы исходной точки геообъекта (для рек, долин и т. д.).

Получение и использование базы данных

Главный веб-сайт базы данных GNIS находится на странице http://geonames.usgs.gov/domestic.

Чтобы скачать национальные географические названия, щелкните по гиперссылке **Download Domestic Names** (Скачать внутренние географические на-

звания), и вам будет предоставлен вариант скачать данные GNIS отдельно для каждого штата либо скачать все объекты одним большим файлом. Можно также скачать «тематические географические справочники», которые содержат избранные подмножества данных – все населенные пункты, все исторические места и т. д. и их координаты.

Если щелкнуть по одной из гиперссылок **File Format** (Формат файла), то появится всплывающее окно, в котором будет приведено более подробное описание структуры файлов.

После скачивания требующихся данных можно просто импортировать файл в базу данных или электронную таблицу. Чтобы импортировать их в электронную таблицу, используйте формат с **разделителями** и введите символ | в качестве пользовательского символа-разделителя. Затем данные можно отсортировать или отфильтровать любым подходящим способом для использования в своем приложении.

Выбор источника геоданных

Если для использования в геоприложениях требуется получить данные географических карт, снимков, высоты объектов или географические названия, то рассмотренные нами источники данных должны предоставить вам все, в чем вы нуждаетесь. Разумеется, это не исчерпывающий список – имеются и другие источники, которые можно найти при помощи онлайновой поисковой системы или сайтов, таких как http://freegis.org.

В следующей ниже таблице перечислены различные требования, которые вы можете предъявить к геоданным при разработке вашего приложения и какой источник(и) данных будет самым подходящим в каждом случае:

Требование	Подходящие источники данных
Простая подложка	Данные границ стран мира World Borders Dataset
Карты со светотеневым изображением рельефа (псевдо 3D)	Данные GLOBE или NED, обработанные при помощи gdaldem; растровые изображения Natural Earth
Карта города	Данные проекта OpenStreetMap
Контуры (план-схемы) города	Данные TIGER (США); городские агломерации Natural Earth
Подробные контуры страны	Географическая база данных береговых линий GSHHG, уровень 1
Фотореалистичные изображения Земли	Снимки проекта Landsat
Список названий городов и других топонимов	Базы данных GNIS (для США) или GEOnet (для остального мира)

Заключение

В этой главе мы провели обзор ряда общедоступных источников геоданных. За данными в векторном формате мы обратились к коллективному сайту OpenStreetMap, где люди могут создать и редактировать векторные карты всего мира, базе

данных TIGER, предоставляемой Бюро переписи населения США в качестве услуги, геоданным веб-сайта Natural Earth, географической базе данных GSHHG высокого разрешения и простому, но эффективному набору данных границ стран мира World Borders Dataset.

За геоданными в растровом формате мы обратились к спутниковым снимкам Landsat, геоданным цифровой модели местности GLOBE и Национальному набору данных рельефа (NED) высокого разрешения для США и его протекторатов.

Затем мы рассмотрели два источника топонимических данных: базу данных сервера географических названий GEOnet, в которой предоставлена информация об официальных географических названиях всех стран, кроме США и за исключением Антарктиды, и данные информационной системы географических названий GNIS, которая предоставляет официальные географические названия в пределах США.

На этом наш обзор источников геоданных завершен. В следующей главе мы начнем работать с инструментами Python, которые были описаны в *главе 3 «Библиотеки Python для геопрограмирования»*, чтобы воспользоваться частью этих геоданных интересным и полезным образом.

Глава 5

Решение задач с геоданными на Python

В этой главе мы соединим рассмотренные нами ранее библиотеки Python и геоданные и воспользуемся ими, чтобы решить ряд задач и получить конкретные результаты. Эти задачи были выбраны таким образом, чтобы продемонстрировать разнообразную методику работы с геоданными в программах на Python; хотя в некоторых случаях существуют методы, которые позволяют достичь этих же результатов быстрее и легче (например, если использовать утилиты командной строки), мы специально создадим эти решения на Python, для того чтобы научиться работать с геоданными в собственных программах на этом языке.

В этой главе будут затронуты следующие темы:

- работа с геоданными в векторном и растровом форматах;
- замена используемых в геоданных датумов и проекций;
- выполнение геопространственных вычислений на точках, линиях и многоугольниках;
- конвертирование и стандартизация единиц геометрии и расстояния.

Эта глава построена как ситуационный справочник, который подробно освещает практические задачи, которые вы, возможно, захотите выполнить, и предоставляет «рецепты» для их выполнения.

Необходимые условия

Если вы хотите выполнить примеры из этой главы, то удостоверьтесь, что на вашем компьютере установлены следующие библиотеки Python:

- пакет динамических библиотек `GDAL/OGR` версии 1.11 или выше (http://gdal.org);
- библиотека Python `pyproj` версии 1.9.4 или выше (http://code.google.com/p/pyproj);
- библиотека Python `Shapely` версии 1.5.9 или выше (https://pypi.python.org/pypi/Shapely).

Для получения дополнительной информации об этих библиотеках и о том, как их применять, пожалуйста, обратитесь к *главе 3 «Библиотеки Python для геопрограммирования»*.

Общие задачи с использованием геоданных

В этом разделе мы рассмотрим несколько примеров задач, которые вы, возможно, захотите выполнить и которые потребуют привлечения геоданных как в векторном, так и в растровом формате.

Задача: вычисление ограничительной рамки для всех стран мира

В этом несколько надуманном примере мы используем файл фигур, чтобы вычислить минимальные и максимальные значения широты и долготы для каждой страны мира. Так называемая «ограничительная рамка», среди прочего, может использоваться для построения карты, центрированной по конкретной стране. Например, ограничительная рамка для Турции будет выглядеть так:

Начните с того, что скачайте набор данных границ стран мира World Borders Dataset с http://thematicmapping.org/downloads/world_borders.php. Удостоверьтесь, что вы скачиваете файл с именем `TM_WORLD_BORDERS-0.3.zip`, а не упрощенный файл `tm_world_borders_simpl-0.3.zip`. Распакуйте архив `.zip` и поместите все файлы, которые составляют файл фигур (файлы `.dbf`, `.prj`, `.shp` и `.shx`), в один подходящий каталог.

Затем нам нужно создать программу на Python, которая сможет прочесть границы каждой страны. К счастью, применение библиотеки `OGR` для чтения содержимого файла фигур достаточно простое:

```
from osgeo import ogr

shapefile = ogr.Open("TM_WORLD_BORDERS-0.3.shp")
```

```
layer = shapefile.GetLayer(0)

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
```

Всю приведенную выше программу вместе со всеми другими примерами можно скачать в составе примеров программ, прилагаемых к данной главе.

Геообъект будет состоять из геометрии, определяющей контур страны, и ряда атрибутов, предоставляющих информацию об этой стране. Согласно файлу `Readme.txt`, атрибуты в файле фигур содержат трехбуквенный код ISO-3166 страны (поле `ISO3`), а также ее название (поле `NAME`), которые позволяют получить код и название страны геообъекта:

```
countryCode = feature.GetField("ISO3")
countryName = feature.GetField("NAME")
```

Кроме того, мы можем получить многоугольник границы страны при помощи метода `GetGeometryRef()`:

```
geometry = feature.GetGeometryRef()
```

Чтобы решить эту задачу, мы должны вычислить ограничительную рамку для этой геометрии. Для обозначения ограничительной рамки геометрии геообъекта в библиотеке `OGR` используется такое понятие, как **огибающая линия**, англ. термин envelope. Так, ограничительную рамку можно получить следующим образом:

```
minLong,maxLong,minLat,maxLat = geometry.GetEnvelope()
```

Соберем все это в полнофункциональную программу:

```
# calcBoundingBoxes.py
from osgeo import ogr

shapefile = ogr.Open("TM_WORLD_BORDERS-0.3.shp")
layer = shapefile.GetLayer(0)

countries = [] # Список кортежей (название, код, мин. шир,
               # макс. шир, мин. долг, макс. долг).

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    countryCode = feature.GetField("ISO3")
    countryName = feature.GetField("NAME")
    geometry = feature.GetGeometryRef()
    minLong,maxLong,minLat,maxLat = geometry.GetEnvelope()
    countries.append((countryName, countryCode,
                      minLat, maxLat, minLong, maxLong))

countries.sort()

for name,code,minLat,maxLat,minLong,maxLong in countries:
    print("{} ({}) lat={:.4f}..{:.4f},long={:.4f}..{:.4f}"
          .format(name,code,minLat,maxLat,minLong,maxLong))
```

Если вы не храните файл фигур TM_WORLD_BORDERS-0.3.shp в том же самом каталоге, что и сам сценарий, то, вызывая ogr.Open(), вам придется добавить строковое значение с месторасположением файла фигур.

В результате выполнения этой программы на выходе получим следующее:

```
% python3 calcBoundingBoxes.py
Afghanistan (AFG) lat=29.4061..38.4721, long=60.5042..74.9157
Albania (ALB) lat=39.6447..42.6619, long=19.2825..21.0542
Algeria (DZA) lat=18.9764..37.0914, long=-8.6672..11.9865
...
```

Задача: вычисление границы между Таиландом и Мьянмой

В этом рецепте мы воспользуемся набором данных границ стран мира, чтобы получить многоугольники, которые описывают границы Таиланда и Мьянмы. Затем мы передадим эти многоугольники библиотеке Shapely и применим ее функционал, чтобы вычислить общую границу между этими двумя странами, сохранив результат в другом файле фигур.

Если у вас нет набора данных границ стран мира, то его следует скачать с вебсайта тематических карт http://thematicmapping.org/downloads/world_ borders.php.

Набор данных границ стран мира к удобству разработчиков включает в себя двухсимвольные коды ISO-3166 стран для каждого объекта, поэтому при чтении файла фигур мы сможем идентифицировать объекты, соответствующие Таиланду и Мьянме. Затем мы можем извлечь контуры всех стран и конвертировать их в объекты геометрии библиотеки Shapely. Вот соответствующий фрагмент, который образует начало нашей программы calcCom-monBorders.py:

```
# calcCommonBorders.py
import os, os.path, shutil
from osgeo import ogr, osr
import shapely.wkt

shapefile = ogr.Open("TM_WORLD_BORDERS-0.3.shp")
layer = shapefile.GetLayer(0)

thailand = None
myanmar = None

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    if feature.GetField("ISO2") == "TH":
        geometry = feature.GetGeometryRef()
        thailand = shapely.wkt.loads(geometry.ExportToWkt())
    elif feature.GetField("ISO2") == "MM":
        geometry = feature.GetGeometryRef()
        myanmar = shapely.wkt.loads(geometry.ExportToWkt())
```


Еще раз отметим, что в этом примере предполагается, что вы поместили файл фигур TM_ WORLD_BORDERS-0.3.shp в тот же самый каталог, что и сценарий Python. Если вы поместили его в другой каталог, то необходимо настроить оператор ogr.Open() так, чтобы учесть месторасположение этого каталога.

Располагая контурами стран в терминах библиотеки Shapely, теперь мы можем воспользоваться ее функционалом вычислительной геометрии, чтобы рассчитать общую границу между этими двумя странами:

```
commonBorder = thailand.intersection(myanmar)
```

В результате получим ломаную (или составную ломаную, в случае если граница разделена на несколько частей). Теперь сохраним эту общую границу в новом файле фигур:

```
if os.path.exists("common-border"):
    shutil.rmtree("common-border")
os.mkdir("common-border")

spatialReference = osr.SpatialReference()
spatialReference.SetWellKnownGeogCS('WGS84')

driver = ogr.GetDriverByName("ESRI Shapefile")
dstPath = os.path.join("common-border", "border.shp")
dstFile = driver.CreateDataSource(dstPath)
dstLayer = dstFile.CreateLayer("layer", spatialReference)

wkt = shapely.wkt.dumps(commonBorder)

feature = ogr.Feature(dstLayer.GetLayerDefn())
feature.SetGeometry(ogr.CreateGeometryFromWkt(wkt))
dstLayer.CreateFeature(feature)
```

Если бы мы показали результаты на карте, то ясно увидели бы общую границу между этими двумя странами:

Мы научимся создавать подобные карты в *главе 7 «Генерирование карт при помощи Python и библиотеки Mapnik»*.

Файл фигур `common-border/border.shp` содержит жирную линию вдоль общей границы стран.

Отметим, что далее в этой главе мы воспользуемся полученным файлом фигур, чтобы вычислить длину границы между Таиландом и Мьянмой, поэтому удостоверьтесь, что вы создали и сохранили копию этого файла фигур.

Задача: анализ высот на основе цифровой карты местности

Файлы с **цифровой картой местности** (ЦКМ), англ. термин Digital Elevation Map (DEM)[1], иногда также именуемые файлами с цифровой сеткой высот, – это растровый формат геоданных, в котором значения каждой ячейки («пиксела») обозначают высоту точки, расположенной на поверхности Земли. Мы встречались с DEM-файлами в предыдущей главе, когда знакомились с двумя экземплярами источников данных, которые предлагают этот тип информации на основе цифровой модели местности (ЦММ): Национальный набор данных рельефа NED, охватывающий только территорию США, и проект GLOBE, который предлагает DEM-файлы с охватом всей Земли.

С учетом того, что DEM-файл содержит данные рельефа, может оказаться интересным проанализировать значения высот для заданного участка. Например, можно начертить гистограмму, показывающую, сколько территории страны находится на определенной высоте. Возьмем немного данных DEM из проекта `GLOBE` и на их основе вычислим гистограмму высот.

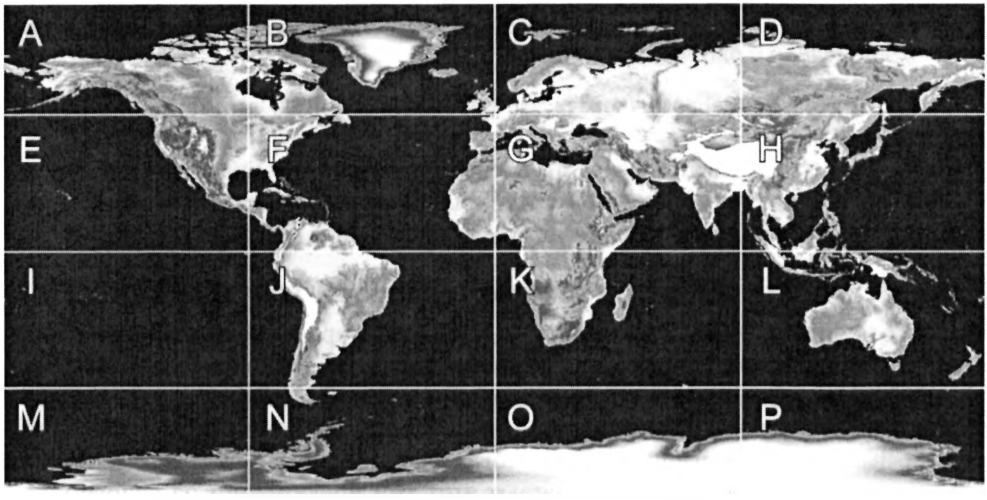
Чтобы не усложнять, выберем Новую Зеландию, то есть небольшую страну, окруженную океаном.

Мы взяли небольшую страну, для того чтобы сократить объем данных для обработки. Кроме того, мы используем страну, окруженную океаном. Это сделано для того, чтобы отметить все точки в пределах ограничительной рамки, не используя для исключения точек, лежащих за пределами границ страны, многоугольник. Это намного упрощает выполнение нашей задачи.

Чтобы скачать данные DEM, зайдите на веб-сайт проекта GLOBE (http://www.ngdc.noaa.gov/mgg/topo/globe.html) и щелкните по гиперссылке **Get Data Online**, чтобы получить онлайновые данные. Мы воспользуемся данными, которые уже были вычислены для этого региона мира, поэтому щелкните по гиперссылке **Any or all 16 «tiles»** для получения какого-то определенного четырехугольного сегмента либо всех 16 сразу. Новая Зеландия находится в четырехугольном сегменте **L** (как показано ниже), поэтому щелкните по гиперссылке для этого сегмента, чтобы его скачать.

---

[1] DEM-файл – формат записи данных высот, представленный регулярной сеткой высотных отметок в трехмерных координатах (xyz) на основе ЦММ. Не путать DEM-файл с цифровой картой местности (ЦКМ) с цифровой моделью местности (ЦММ). Последняя является более общим понятием. – *Прим. перев.*

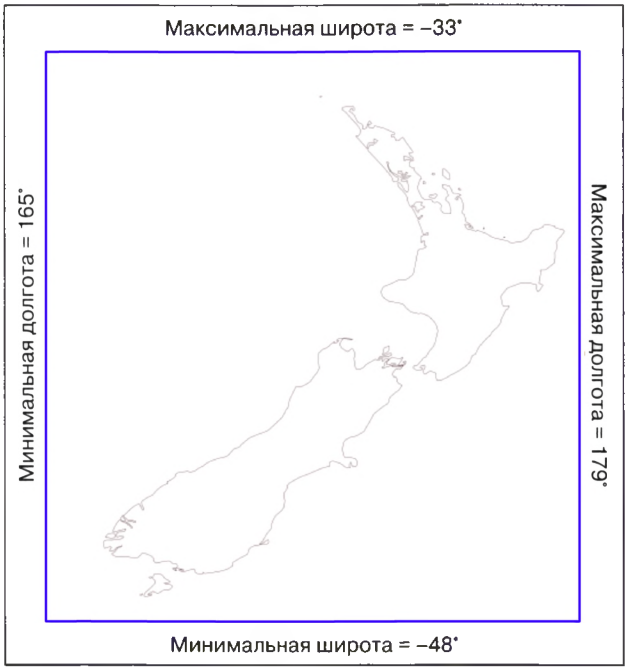


Скачиваемый вами файл называется `l10g.zip` (или `l10g.gz`, если вы приняли решение скачать сегмент в формате `gzip`). Если вы распакуете его, то у вас получится один файл с именем `l10g`, содержащий исходные высотные данные.

Сам по себе этот файл мало полезен – чтобы можно было сопоставить каждое значение высоты с его позицией на Земле, он должен быть привязан к поверхности Земли. Для этого необходимо скачать связанный с ним заголовочный файл. Подходящие заголовочные файлы для предварительно сгенерированных сегментов находятся на странице http://www.ngdc.noaa.gov/mgg/topo/elev/esri/hdr. Скачайте файл с именем `l10g.hdr` и поместите его в тот же самый каталог, что и файл `l10g`, который вы скачали ранее. Затем файл DEM можно прочесть при помощи библиотеки `GDAL` точно так же, как мы делали это ранее:

```
from osgeo import gdal
dataset = gdal.Open("l10g")
```

Вы, вероятно, заметили, когда скачали сегмент `l10g`, что он охватывает территорию, намного большую, чем просто Новую Зеландию, – включена вся Австралия, а также Малайзия, Папуа – Новая Гвинея и несколько других восточноазиатских стран. Чтобы работать с высотными данными, относящимися только к Новой Зеландии, мы должны идентифицировать соответствующий участок растрового DEM-файла, то есть диапазон прямоугольных координат x и y, который охватывает Новую Зеландию. Начнем с того, что посмотрим на карту и определим минимальные и максимальные значения широты и долготы, которые окаймляют всю Новую Зеландию и никакую другую страну:



Округлив до ближайшего градуса, мы получим ограничительную рамку в виде значений долготы и широты (165, –48)...(179, –33). Это именно та область, которую мы хотим просканировать, чтобы охватить всю Новую Зеландию.

Однако здесь имеется одно затруднение: растровые данные состоят из пикселов, или «ячеек», идентифицируемых прямоугольными координатами x и y, а не значениями долготы и широты. Мы должны перевести долготы и широты в прямоугольные координаты x и y. Для этого нам нужно воспользоваться имеющимся в библиотеке GDAL функционалом трансформирования координат.

Начнем с того, что получим геотрансформацию нашего набора данных:

```
t = dataset.GetGeoTransform()
```

Используя эту геотрансформацию, мы можем перевести прямоугольную координату (x, y) в соответствующее ей значение широты и долготы. Впрочем, сейчас мы хотим сделать противоположное – взять широту и долготу и вычислить соответствующую прямоугольную координату x и y. Для этого нужно *инвертировать* геотрансформацию. И снова библиотека GDAL сделает это за нас:

```
success,tInverse = gdal.InvGeoTransform(t)
if not success:
    print("Неудачно!")
    sys.exit(1)
```

Бывают случаи, когда преобразование не может быть инвертировано. Именно поэтому метод `gdal.InvGeoTransform()` вместе с результатом обратной трансформации возвращает флаг `success`, обозначающий успешность выполнения операции. Впрочем, в случае с используемыми данными DEM-файла трансформация должна всегда быть обратимой, и поэтому это вряд ли случится.

Имея обратную геотрансформацию, теперь можно перевести широту и долготу в прямоугольную координату x и y, как показано ниже:

```
x,y = gdal.ApplyGeoTransform(tInverse, longitude, latitude)
```

С их помощью мы, наконец, можем определить минимальные и максимальные прямоугольные координаты (x, y), которые охватывают интересующую нас область:

```
x1,y1 = gdal.ApplyGeoTransform(tInverse, minLong, minLat)
x2,y2 = gdal.ApplyGeoTransform(tInverse, maxLong, maxLat)

minX = int(min(x1, x2))
maxX = int(max(x1, x2))
minY = int(min(y1, y2))
maxY = int(max(y1, y2))
```

Зная прямоугольные координаты x и y интересующего нас участка ЦКМ, мы можем воспользоваться библиотекой `GDAL`, чтобы прочитать отдельные значения высоты. Начнем с того, что получим растровый канал, который содержит данные ЦКМ:

```
band = dataset.GetRasterBand(1)
```

Каналы в библиотеке `GDAL` нумеруются с 1. В используемых нами данных ЦКМ имеется всего 1 канал.

Затем мы прочитаем значения высот из нашего растрового канала. Напомним, что существуют два метода чтения растровых данных при помощи библиотеки `GDAL`: чтение исходных двоичных данных, которые затем извлекаются при помощи модуля `struct`, либо при помощи библиотеки `NumPy` для автоматического считывания данных в массив.

Предположим, что библиотека `NumPy` не установлена; вместо этого воспользуемся методом `struct`. Используя прием, который обсуждался в *главе 3 «Библиотеки Python для геопрограммирования»*, для чтения исходных двоичных данных можно воспользоваться методом `ReadRaster()` и затем функцией `struct.unpack()`, чтобы конвертировать двоичные данные в список значений высот. Вот соответствующий фрагмент программы:

```
fmt = "<" + ("h" * width)
scanline = band.ReadRaster(X, y, width, height,
                           width, height, gdalconst.GDT_Int16)
values = struct.unpack(fmt, scanline)
```

Собрав все это вместе, мы сможем при помощи библиотеки GDAL открыть растровый файл данных и прочесть все значения высот в пределах окружающей Новую Зеландию ограничительной рамки. Далее мы рассчитаем частотность появления каждого значения высоты и распечатаем простую гистограмму, используя значения высоты. Вот результирующая программа:

```
# histogram.py

import sys, struct
from osgeo import gdal
from osgeo import gdalconst

minLat = -48
maxLat = -33
minLong = 165
maxLong = 179

dataset = gdal.Open("110g")
band = dataset.GetRasterBand(1)

t = dataset.GetGeoTransform()
success,tInverse = gdal.InvGeoTransform(t)
if not success:
    print("Неудача!")
    sys.exit(1)

x1,y1 = gdal.ApplyGeoTransform(tInverse, minLong, minLat)
x2,y2 = gdal.ApplyGeoTransform(tInverse, maxLong, maxLat)

minX = int(min(x1, x2))
maxX = int(max(x1, x2))
minY = int(min(y1, y2))
maxY = int(max(y1, y2))

width = (maxX - minX) + 1
fmt = "<" + ("h" * width)

histogram = {}  # Ставит высоту в соответствие частотности.

for y in range(minY, maxY+1):
    scanline = band.ReadRaster(minX, y, width, 1,
                               width, 1,
                               gdalconst.GDT_Int16)
  values = struct.unpack(fmt, scanline)

  for value in values:
      try:
          histogram[value] += 1
      except KeyError:
          histogram[value] = 1

for height in sorted(histogram.keys()):
    print(height, histogram[height])
```

Напомним, что в случае если вы поместили файл `110g` в другой каталог, то следует добавить его адрес на диске в оператор `gdal.Open()`.

Если вы выполните этот пример, то увидите список высот (в метрах) и количество ячеек, которые имеют такое же значение высоты:

```
-500 2607581
1 6641
2 909
3 1628
...
3097 1
3119 2
3173 1
```

Результат обнаруживает еще одну, заключительную, проблему: наличие большого количества ячеек со значением высоты, равным `-500`. В чем же тут дело? Ясно, что значение `-500` недопустимо для высоты.

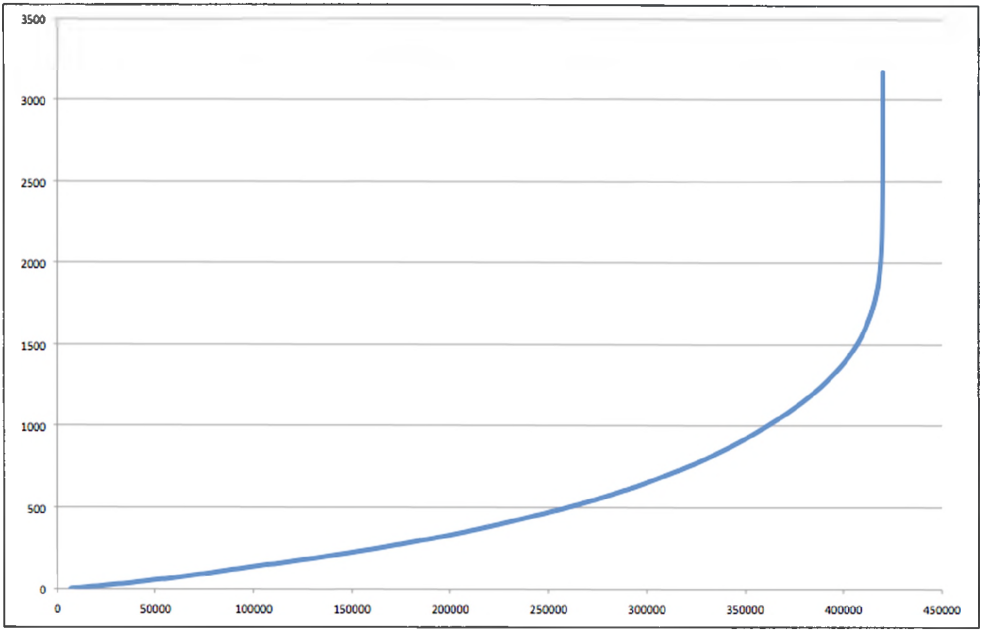
Документация по GLOBE дает ответ:

> *«Все сегменты содержат значения –500 для океанов, причем значения между –500 и минимальным значением для суши здесь не отмечены».*

Таким образом, все точки со значением `-500` обозначают ячейки, которые расположены над океаном. Если вы помните из нашего описания библиотеки `GDAL` в *главе 3 «Библиотеки Python для геопрограммирования»*, библиотека `GDAL` располагает специальным значением, предназначенным для отсутствующих данных **no data**, которое существует именно для таких целей. И действительно, объект `gdal.RasterBand` имеет метод `GetNoDataValue()`, который сообщает, какое это значение. Этот факт позволяет добавить в нашу программу строку, выделенную полужирным шрифтом, которая даст возможность исключить ячейки с отсутствующими данными:

```
for value in values:
    if value != band.GetNoDataValue():
        try:
            histogram[value] += 1
        except KeyError:
            histogram[value] = 1
```

В результате мы, наконец, получим гистограмму высот по всей Новой Зеландии. На основе этих данных можно, если понадобится, построить график. Например, следующая диаграмма показывает итоговое количество пикселов, которые расположены не выше заданной высоты:



Смена датумов и проекций

В *главе 2 «Геоинформационные системы»* мы узнали, что **датум** – это математическая модель фигуры Земли, тогда как **проекция** – метод трансляции точек на поверхности Земли в точки на двумерной карте. Существует огромное количество датумов и проекций – во время работы с геоданными всегда нужно знать используемые в них датум и проекцию (если таковые имеются). Если вы совмещаете данные из нескольких источников, то нередко приходится вносить в геоданные изменения, переводя их из одного датума в другой или из одной проекции в другую.

Задача: смена проекции для совмещения файлов фигур с географическими и UTM-координатами

В этом рецепте мы обратимся к двум файлам фигур с разными картографическими проекциями. Нам еще не встречались геоданные, в которых использовалась бы проекция, – во всех данных, которые мы встречали до сих пор, использовались неспроецированные значения широты и долготы. Поэтому начнем с того, что скачаем немного геоданных в **проекции Меркатора (UTM)**.

Веб-сайт WebGIS (http://www.webgis.com) предоставляет данные в виде файлов фигур с описанием землепользования и ландшафта под названием LULC (аббревиатура Land Use and Land Cover). Для нашего примера мы скачаем файл фигур, в котором используется проекция UTM с охватом южной Флориды (точнее, округа Дейд).

Этот файл фигур можно скачать по прямой ссылке http://www.webgis.com/MAPS/fl/lulcutm/miami.zip.

Распакованная папка содержит файл фигур `miami.shp` и файл `da-tum_reference.txt`, в котором описывается система координат файла фигур.

В нем (в переводе) говорится следующее:

```
Файл фигур LULC получен из исходного файла USGS GIRAS LULC
компанией Lakes Environmental Software.
Датум: NAD83
Проекция: UTM
Зона: 17
Дата сбора данных Геологической службой США (ГС): 1972
Ссылка: http://edcwww.cr.usgs.gov/products/landcover/lulc.html
```

Итак, в этом файле фигур используются 17-я зона проекции UTM и датум NAD83.

Теперь возьмем второй файл фигур, на этот раз в географических координатах. Для этого воспользуемся базой данных береговых линий GSHHS, в которой используются датум WGS84 и географические координаты (широты и долготы).

Для этого примера не нужно скачивать всю базу данных GSHHS; несмотря на то что мы изобразим карту с наложением данных LULC поверх данных GSHHS, вам, чтобы закончить этот рецепт, потребуется только файл фигур LULC. Рисование карт, аналогичных рассмотренному в этом рецепте, будет обсждаться в *главе 7 «Генерирование карт при помощи Python и библиотеки Mapnik»*.

Нам не удастся сравнить координаты этих двух файлов фигур напрямую; файл фигур LULC имеет координаты, измеряемые в единицах UTM (то есть в метрах от заданной линии отсчета), тогда как файл фигур GSHHS имеет координаты в значениях широты и долготы (в десятичных градусах):

```
LULC:  x=485719.47, y=2783420.62
       x=485779.49, y=2783380.63
       x=486129.65, y=2783010.66
       ...
GSHHS: x=180.0000, y=68.9938
       x=180.0000, y=65.0338
       x=179.9984, y=65.0337
```

Прежде чем мы сможем эти два файла фигур совместить, нам сначала нужно их конвертировать, с тем чтобы использовать одинаковую проекцию. Мы сделаем это путем конвертирования файла фигур LULC из координат UTM-17 в географические координаты (широты и долготы). Для этого нам потребуется задать **трансформацию координат** в терминах библиотеки `OGR` и затем применить эту трансформацию ко всем объектам в файле фигур.

Ниже показано, каким образом определить трансформацию координат при помощи библиотеки `OGR`:

```
from osgeo import osr

srcProjection = osr.SpatialReference()
srcProjection.SetUTM(17)

dstProjection = osr.SpatialReference()
dstProjection.SetWellKnownGeogCS('WGS84') # Широта/Долгота

transform = osr.CoordinateTransformation(srcProjection,
                                         dstProjection)
```

На основе этой трансформации мы сможем перевести все геообъекты в файле фигур из координат UTM назад в географические координаты:

```
for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    geometry = feature.GetGeometryRef()
    geometry.Transform(transform)
```

Собрав все вместе и добавив ранее рассмотренные методы копирования геообъектов из одного файла фигур в другой, мы получим следующую ниже полнофункциональную программу:

```
# changeProjection.py

import os, os.path, shutil
from osgeo import ogr
from osgeo import osr
from osgeo import gdal

# Задать исходную и целевую проекции, а также объект трансформации
# координат для конвертирования из одной проекции в другую.

srcProjection = osr.SpatialReference()
srcProjection.SetUTM(17)

dstProjection = osr.SpatialReference()
dstProjection.SetWellKnownGeogCS('WGS84') # Lat/long.

transform = osr.CoordinateTransformation(srcProjection,
                                         dstProjection)

# Открыть исходный файл фигур.

srcFile = ogr.Open("miami/miami.shp")
srcLayer = srcFile.GetLayer(0)

# Создать целевой файл фигур dest и назначить ему новую проекцию.

if os.path.exists("miami-reprojected"):
    shutil.rmtree("miami-reprojected")
os.mkdir("miami-reprojected")

driver = ogr.GetDriverByName("ESRI Shapefile")
dstPath = os.path.join("miami-reprojected", "miami.shp")
dstFile = driver.CreateDataSource(dstPath)
```

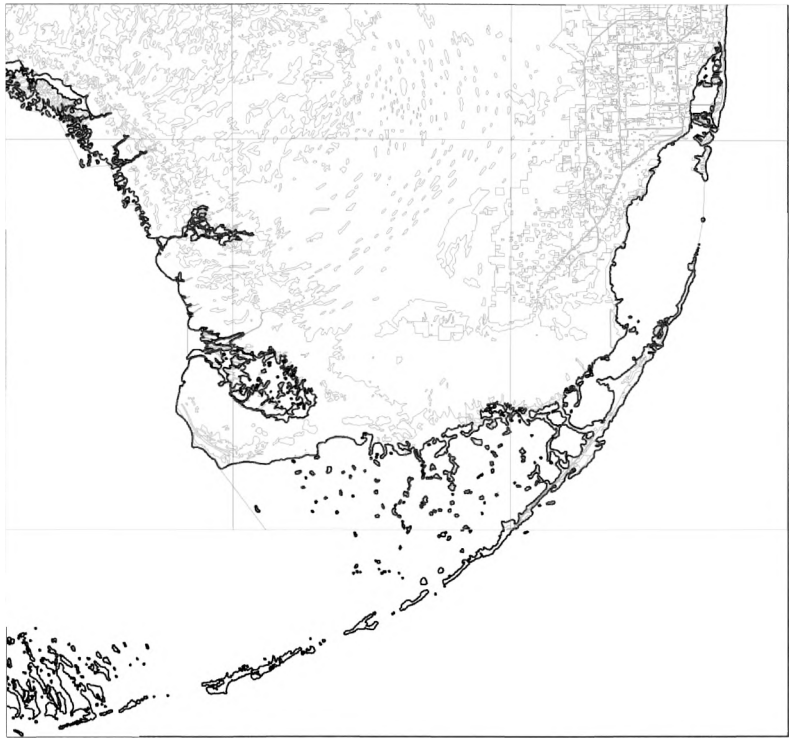
```
dstLayer = dstFile.CreateLayer("layer", dstProjection)

# Последовательно перепроецировать все геообъекты.

for i in range(srcLayer.GetFeatureCount()):
    feature = srcLayer.GetFeature(i)
    geometry = feature.GetGeometryRef()
    newGeometry = geometry.Clone()
    newGeometry.Transform(transform)
    feature = ogr.Feature(dstLayer.GetLayerDefn())
    feature.SetGeometry(newGeometry)
    dstLayer.CreateFeature(feature)
```

Отметим, что этот пример не копирует значения полей в новый файл фигур; если ваш файл фигур имеет метаданные, то вам придется переносить поля при создании каждого нового геообъекта. Кроме того, приведенный выше пример предполагает, что файл фигур `miami.shp` помещен в подкаталог `miami`; необходимо изменить оператор `ogr.Open()` таким образом, чтобы в нем учитывался соответствующий адрес каталога, в случае если вы сохранили этот файл фигур в другом месте.

После выполнения этой программы с файлом фигур `miami.shp` координаты для всех объектов в файле фигур будут переведены из координат UTM-17 в географические координаты:

```
До перепроецирования:    x=485719.47, y=2783420.62
                         x=485779.49, y=2783380.63
                         x=486129.65, y=2783010.66
                         ...
После перепроецирования: x=-81.1417, y=25.1668
                         x=-81.1411, y=25.1664
                         x=-81.1376, y=25.1631
```

Чтобы выяснить, что программа работает как надо, составим карту, изображающую перепроецированные данные LULC с наложением поверх данных береговой линии GSHHS:



Светло-серые контуры показывают различные многоугольники в файле фигур LULC, а черный контур – береговую линию в соответствии с их определением в файле фигур GLOBE. В обоих файлах фигур теперь используются географические координаты и, как можете убедиться, соответствие береговых линий точное.

Если вы были внимательны, то, возможно, заметили, что в данных LULC используется датум NAD83, тогда как в данных GSHHS (береговые линии) и в нашей перепроецированной версии данных LULC используется датум WGS84. Преобразование выполняется безошибочно, потому что эти два датума идентичны для точек в пределах территории Северной Америки.

Задача: перевод из одного датума в другой для совмещения свежих данных TIGER со старыми

Для этого примера мы должны получить немного геоданных, в которых используется датум NAD27. Этот датум, относящийся к 1927 году, вплоть до 1980-х годов обычно использовался для геоанализа североамериканского континента и затем был заменен на датум NAD83.

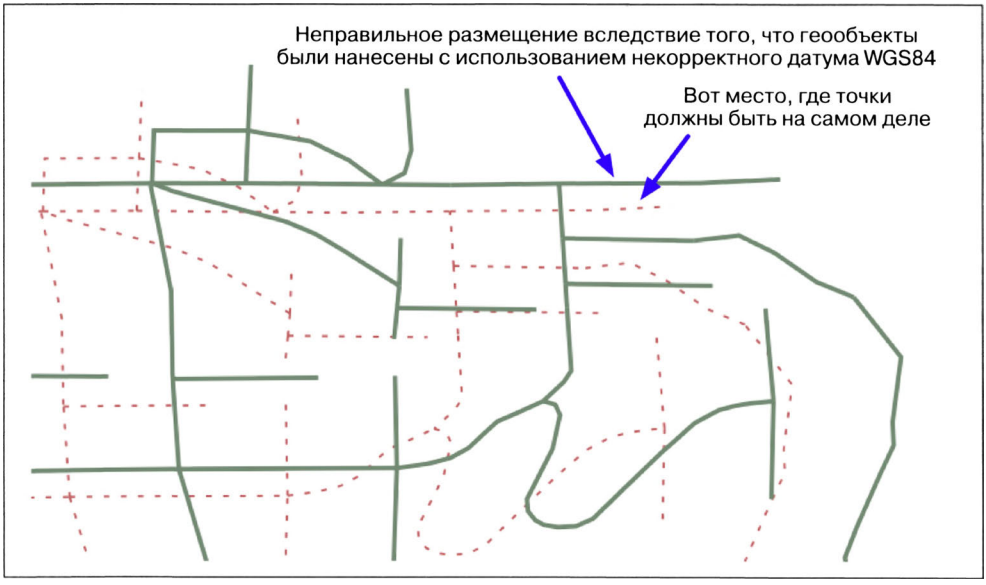
Геологическая служба США предоставляет набор данных транспортной сети, в котором используется предыдущий датум NAD27. Чтобы получить эти данные, пройдите на http://pubs.usgs.gov/dds/dds-81/TopographicData/Roads и скачайте файлы `roads.dbf`, `roads.prj`, `roads.shp` и `roads.shx`. Эти четыре файла составляют файл фигур дорожной сети «roads». Разместите их в каталоге с именем `roads`.

Этот демонстрационный файл фигур охватывает район гор Сьерра-Невады в США, включая часть шт. Калифорния и шт. Невада. Этот файл фигур также охватывает г. Бриджпорт, шт. Калифорния.

При помощи инструмента командной строки `ogrinfo` мы можем убедиться, что этот файл фигур действительно использует предыдущий датум NAD27:

```
% ogrinfo -so roads.shp roads
INFO: Open of `roads.shp'
      using driver `ESRI Shapefile' successful.
Layer name: roads
Geometry: Line String
Feature Count: 10050
Extent: (-119.637220, 37.462695) - (-117.834877, 38.705226)
Layer SRS WKT:
GEOGCS["GCS_North_American_1927",
    DATUM["North_American_Datum_1927",
        SPHEROID["Clarke_1866",6378206.4,294.9786982]],
    PRIMEM["Greenwich",0],
    UNIT["Degree",0.0174532925199433]]
...
```

Если бы, допустим, в этом файле фигур использовался более распространенный датум WGS84, то все геообъекты появились бы в неправильных местах:



Чтобы они появились в правильных местах и чтобы совместить эти геообъекты с другими, которые основаны на датуме WGS84, мы должны конвертировать файл фигур так, чтобы можно было использовать датум WGS84. Перевод файла фигур из одного датума в другой требует того же самого базового процесса, который мы применили ранее с целью перевода файла фигур из одной проекции в другую.

Прежде всего вы выбираете исходный и целевой датумы и задаете трансформацию координат, чтобы конвертировать их из одной системы в другую:

```
srcDatum = osr.SpatialReference()
srcDatum.SetWellKnownGeogCS('NAD27')

dstDatum = osr.SpatialReference()
dstDatum.SetWellKnownGeogCS('WGS84')

transform = osr.CoordinateTransformation(srcDatum, dstDatum)
```

Затем обрабатываете каждый объект в файле фигур, трансформируя геометрию геообъекта при помощи процедуры трансформации координат:

```
for i in range(srcLayer.GetFeatureCount()):
    feature = srcLayer.GetFeature(i)
    geometry = feature.GetGeometryRef()
    geometry.Transform(transform)
```

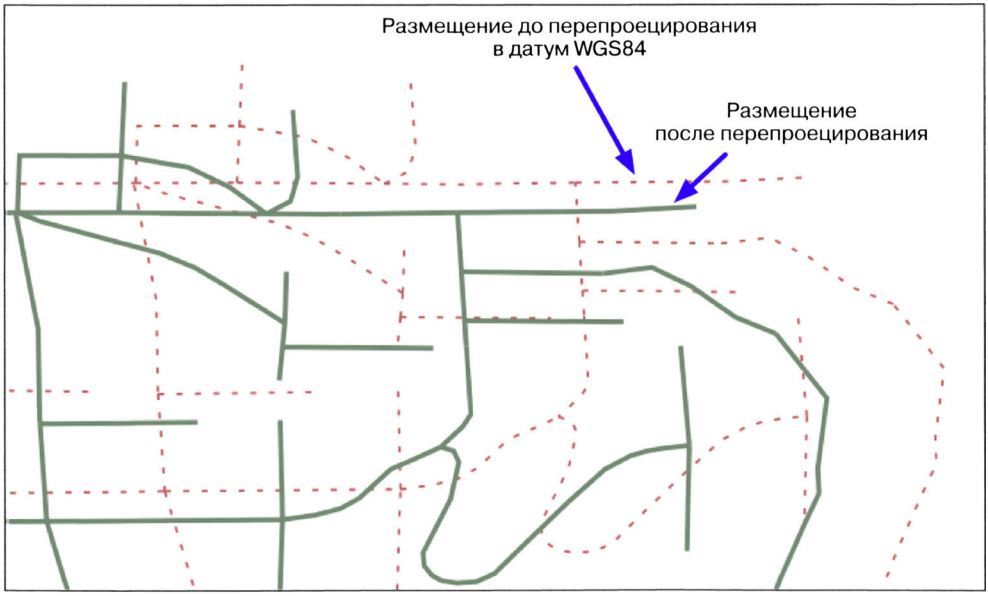
Вот полнофункциональная программа на Python для конвертирования файла фигур roads из датума NAD27 в датум WGS84:

```
# changeDatum.py

import os, os.path, shutil
from osgeo import ogr
from osgeo import osr
from osgeo import gdal

# Задать исходный и целевой датумы и объект трансформации
# координат для конвертирования из одного датума в другой.

srcDatum = osr.SpatialReference()
srcDatum.SetWellKnownGeogCS('NAD27')

dstDatum = osr.SpatialReference()
dstDatum.SetWellKnownGeogCS('WGS84')

transform = osr.CoordinateTransformation(srcDatum, dstDatum)

# Открыть исходный файл фигур.

srcFile = ogr.Open("roads/roads.shp")
srcLayer = srcFile.GetLayer(0)

# Создать целевой файл фигур dest и назначить ему новую проекцию.

if os.path.exists("roads_reprojected"):
    shutil.rmtree("roads_reprojected")
os.mkdir("roads_reprojected")
```

```
driver = ogr.GetDriverByName("ESRI Shapefile")
dstPath = os.path.join("roads_reprojected", "roads.shp")
dstFile = driver.CreateDataSource(dstPath)
dstLayer = dstFile.CreateLayer("layer", dstDatum)

# Последовательно перепроецировать все геообъекты.

for i in range(srcLayer.GetFeatureCount()):
    feature = srcLayer.GetFeature(i)
    geometry = feature.GetGeometryRef()
    newGeometry = geometry.Clone()
    newGeometry.Transform(transform)
    feature = ogr.Feature(dstLayer.GetLayerDefn())
    feature.SetGeometry(newGeometry)
    dstLayer.CreateFeature(feature)
```

Приведенная выше программа предполагает, что папка `roads` находится в том же самом каталоге, что и сам сценарий Python; если вы поместили эту папку где-то в другом месте, то необходимо изменить оператор `ogr.Open()`, чтобы указать соответствующий путь к каталогу.

Если после перепроецирования в датум WGS84 теперь визуализировать геообъекты снова, то они появятся в нужных местах:



Выполнение геопространственных расчетов

Библиотека Shapely очень эффективно выполняет различные расчеты на геоданных. Проверим ее работу в деле на примере решения сложной практической задачи.

Задача: идентификация национальных парков внутри и в окрестностях городских агломераций

Бюро переписи населения США предоставляет файл фигур, содержащий так называемые **статистические ареалы вокруг городского центра**, англ. термин Core Based Statistical Areas (CBSA), то есть многоугольники, обозначающие городские агломерации, состоящие из одного или нескольких районов, опирающихся на единый городской центр с населением 10 000 человек и выше, плюс окрестные районы, которые связаны с центром в социально-экономическом плане. В то же время веб-сайт GNIS предоставляет топонимические списки и другую информацию. Используя эти два источника данных, мы определим национальные парки в пределах городской агломерации или в близлежащих от нее районах.

Из-за объема данных, с которыми мы имеем дело, мы ограничим наш поиск шт. Калифорния. Чтобы проверить все комбинации многоугольник–топоним в статистических ареалах вокруг городского центра на всей территории США, потребовалось бы очень продолжительное время; можно оптимизировать программу, чтобы она выполняла расчет быстро, однако оптимизация сделала бы этот пример слишком сложным для наших текущих целей.

Начнем с того, что скачаем необходимые данные. Скачаем файл фигур, содержащий все статистические ареалы США вокруг городского центра. Перейдите на страницу TIGER на http://census.gov/geo/www/tiger. Нажмите на ссылку **TIGER/Line Shapefiles** (Файлы фигур с данными в формате TIGER/Line), затем перейдите в раздел **Download** (Скачать), чтобы скачать последнюю версию файлов фигур **TIGER/Line** (на момент перевода книги это была версия 2016 года). Выберите опцию **FTP Interface** (Передача по протоколу FTP), и ваш компьютер откроет сеанс FTP для соответствующего каталога. Зарегистрируйтесь как гость, перейдите в каталог `CBSA` и скачайте соответствующий файл, который там находится.

Если не получается попасть на веб-сайт Бюро переписи США обычным образом, то воспользуйтесь анонимайзером Хамелеон (http://cameleo.ru/) и перейдите на нужную страницу 2016 TIGER/Line® Shapefiles: Core Based Statistical Areas (Файлы фигур TIGER/Line за 2016 г.: статистические ареалы вокруг городского центра) по прямой ссылке http://o53xo.mnsw443vomxgo33w.cmle.ru/cgi-bin/geo/shapefiles/index.php?year=2016&layergroup=Core+Based+Statistical+Areas, затем нажмите на первой гиперссылке Metropolitan/Micropolitan Statistical Area (Метрополисные/микрополисные статистические ареалы).

Нужный вам файл имеет имя `tl_XXXX_us_cbsa.zip`, где `XXXX` – это год создания файла данных. После того как вы скачаете файл, распакуйте его и поместите получившийся файл фигур в удобное место на диске, с тем чтобы можно было с ним работать.

Теперь необходимо загрузить данные географических названий GNIS. Пройдите на веб-сайт GNIS http://geonames.usgs.gov/domestic и щелкните по гиперссылке **Download Domestic Names** (Скачать внутренние географические назва-

ния), чтобы скачать национальные географические названия. Так как нам нужны только данные по Калифорнии, то выбираем этот штат из всплывающего меню **Select state for download** (Выбрать штат для скачивания) со списком штатов для скачивания, которое расположено в разделе **States, Territories, Associated Areas of the United States** страницы, посвященном штатам, территориям и ассоциированным с США регионам. В результате вы получите файл с именем `CA_Features_XXXX.txt`, где `XXXX` – это дата создания файла. Поместите этот файл и полученный ранее файл фигур с CBSA в какое-нибудь удобное для вас место.

Теперь все готово, чтобы приступить к написанию нашей программы. Начнем с того, что прочитаем файл фигур статистических ареалов вокруг городского центра (CBSA) и извлечем многоугольники, которые описывают контур каждой городской агломерации в виде объекта геометрии библиотеки `Shapely`:

```
shapefile = ogr.Open("tl_2015_us_cbsa.shp")
layer = shapefile.GetLayer(0)

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    geometry = feature.GetGeometryRef()
    wkt = geometry.ExportToWkt()
    outline = shapely.wkt.loads(wkt)
```

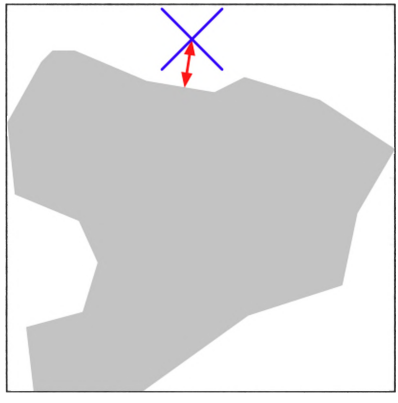
Затем нам нужно просмотреть файл `CA_Features_XXX.txt`, чтобы определить объекты, маркированные как национальные парки. Из каждого такого геообъекта извлечем его название и связанные с ним широту и долготу. Вот как это может выглядеть:

```
f = open("CA_Features_XXX.txt", "r")
for line in f.readlines():
    chunks = line.rstrip().split("|")
    if chunks[2] == "Park":
      name = chunks[1]
      latitude = float(chunks[9])
      longitude = float(chunks[10])
```

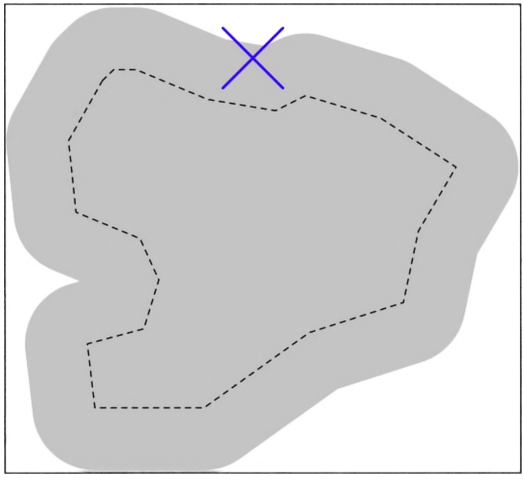
Напомним, что база данных географических названий GNIS – это текстовый файл с разделением полей данных символом вертикальной черты, и поэтому мы должны разделить строку при помощи конструкции `line.rstrip().split("|")`.

А теперь наступает самая интересная часть: мы должны выяснить, какие национальные парки лежат в пределах или поблизости от каждой городской агломерации. Это можно сделать двумя способами, причем любой из них будет работать:

- можно воспользоваться методом `outline.distance()`, чтобы вычислить расстояние между контуром и объектом `Point`, обозначающим географическое положение национального парка:



- можно *расширить* многоугольник, используя метод `outline.buffer()`, и затем проверить, содержит ли получившийся многоугольник заданную точку:



При обработке большого количества точек второй вариант работает быстрее, так как мы можем вычислить расширенные многоугольники заранее и затем использовать их для последовательного сопоставления с каждой точкой. Остановимся на этом варианте:

```
# findNearbyParks.py
from osgeo import ogr
import shapely.geometry
import shapely.wkt

MAX_DISTANCE = 0.1 # Угловое расстояние; приб. 10 км.

print("Загрузка городских агломераций...")
```

```
urbanAreas = {} # Ставит название агломерации в соответствие
                # многоугольнику Shapely

shapefile = ogr.Open("tl_2015_us_cbsa.shp")
layer = shapefile.GetLayer(0)

for i in range(layer.GetFeatureCount()):
    print("Расширение объекта {} из {}"
          .format(i, layer.GetFeatureCount()))
    feature = layer.GetFeature(i)
    name = feature.GetField("NAME")
    geometry = feature.GetGeometryRef()
    outline = shapely.wkt.loads(geometry.ExportToWkt())
    dilatedOutline = outline.buffer(MAX_DISTANCE)
    urbanAreas[name] = dilatedOutline

print("Проверка национальных парков...")

f = open("CA_Features_XXX.txt", "r")
for line in f.readlines():
    chunks = line.rstrip().split("|")
    if chunks[2] == "Park":
        parkName = chunks[1]
        latitude = float(chunks[9])
        longitude = float(chunks[10])
        pt = shapely.geometry.Point(longitude, latitude)
        for urbanName,urbanArea in urbanAreas.items():
            if urbanArea.contains(pt):
                print("{} находится в пределах или поблизости от {}".format(parkName,
urbanName))
f.close()
```

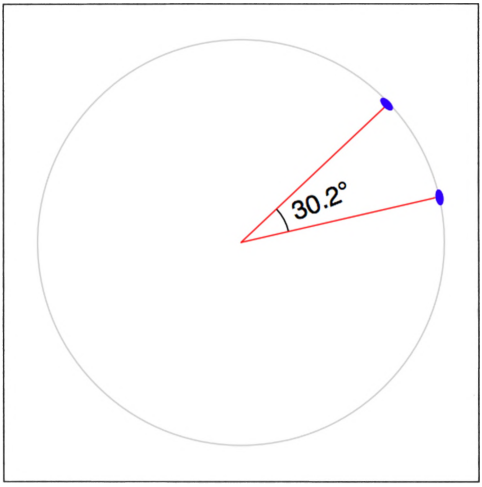
Напомним, что нужно поменять имя файла `CA_Features_XXXX.txt`, чтобы он соответствовал фактическому имени скачанного вами файла. Кроме того, в программе следует добавить путь к ссылкам на файлы `tl_2015_us_cbsa.shp` и `CA_Features_XXXX.txt`, в случае если вы поместили их в другой каталог.

На выполнение этой программы потребуется несколько минут, после чего вы получите полный список всех национальных парков, которые находятся в пределах или поблизости от городской агломерации:

```
% python3 findNearbyParks.py
Выполняется загрузка городских агломераций...
Выполняется проверка национальных парков...
Imperial National Wildlife Refuge is in or near El Centro, CA
Imperial National Wildlife Refuge is in or near Yuma, AZ
Cibola National Wildlife Refuge is in or near El Centro, CA
Twin Lakes State Beach is in or near Santa Cruz-Watsonville, CA
Admiral William Standley State Recreation Area is in or near Ukiah, CA
...
```

Работа нашей программы занимает много времени, потому что она содержит городские агломерации по всем США, а не только одного шт. Калифорния. Очевидно, оптимизация прежде всего должна заключаться во включении только городских агломераций шт. Калифорния, что значительно ускорит работу программы.

Отметим, что, для того чтобы решить, не находится ли национальный парк внутри или рядом с конкретной городской агломерацией, наша программа использует **угловые расстояния**. Как уже упомяналось в *главе 2 «Геоинформационные системы»*, угловое расстояние – это угол между двумя лучами (сторонами угла), исходящими из центра Земли к ее поверхности:

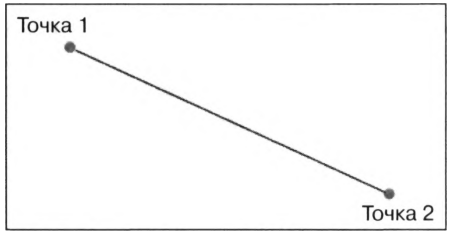


Так как мы имеем дело с данными для шт. Калифорния, где 1° угловой меры приблизительно равняется 100 км на поверхности Земли, то угловая мера в 0.1° будет примерно равна действительному расстоянию 10 км.

Ясно, что использование угловых мер упрощает и облегчает расчет расстояния, правда, при этом он не дает точного расстояния на поверхности Земли. Если от вашего приложения требуется расчет точного расстояния, то угловое расстояние было бы целесообразно использовать в самом начале с целью фильтрации геообъектов, которые очевидным образом находятся слишком далеко, и затем получить точный результат по оставшимся объектам путем вычисления точки на границе многоугольника, самой близкой к заданной точке, и вычисления линейного расстояния между двумя точками. И тогда вы отбросите те точки, которые превышают ваше заданное точное линейное расстояние. Реализация этого алгоритма стала бы интересным испытанием, и все-таки не тем, чем мы будем заниматься в оставшейся части этой книги.

Конвертирование и стандартизация единиц геометрии и расстояния

Предположим, что у вас есть две точки на поверхности Земли, и между ними проведен отрезок прямой:



Каждую из этих точек можно описать как координату, используя для этого некоторую произвольную систему координат (например, при помощи значений широты и долготы), тогда как длину отрезка прямой можно рассматривать как расстояние между двумя точками.

Разумеется, из-за того, что поверхность Земли не плоская, мы на самом деле имеем дело вовсе не с отрезками прямых, а вычисляем геодезические или ортодромические расстояния, т. е. **по дуге большого круга** на поверхности Земли.

При наличии любых двух координат можно вычислить между ними расстояние. С другой стороны, можно начать с одной координаты, заданого расстояния и направления и затем вычислить координаты другой точки.

Библиотека Python `pyproj` позволяет выполнять подобные типы вычислений для любого нужного датума. Библиотеку `pyproj` можно также использовать для перевода из прямоугольных координат назад в географические координаты и наоборот, причем она дает вам возможность выполнять эти виды расчетов для любого конкретного датума, системы координат и картографической проекции.

В конечном счете такая геометрия, как ломаная или многоугольник, состоит всего лишь из списка связанных точек. Это означает, что, используя процесс, который мы описали ранее, можно вычислить геодезическое расстояние между любой связанной парой точек в многоугольнике или ломаной и просуммировать результаты для получения фактической длины любой геометрии. Применим эти знания, чтобы решить практическую задачу.

Задача: вычисление длины границы между Тайландом и Мьянмой

Чтобы решить эту задачу, мы воспользуемся созданным ранее файлом фигур `common-border/border.shp`. Этот файл фигур содержит единственный объект – ломаную (или последовательность отрезков, что то же самое), которая описывает

границу между этими двумя странами. Начнем с того, что рассмотрим отдельные отрезки ломаной, которые образуют геометрию этого объекта:

```
import os.path
from osgeo import ogr

def getLineSegmentsFromGeometry(geometry):
    segments = []
    if geometry.GetPointCount() > 0:
        segment = []
        for i in range(geometry.GetPointCount()):
            segment.append(geometry.GetPoint_2D(i))
        segments.append(segment)
    for i in range(geometry.GetGeometryCount()):
        subGeometry = geometry.GetGeometryRef(i)
        segments.extend(
            getLineSegmentsFromGeometry(subGeometry))
    return segments

filename = os.path.join("common-border", "border.shp")
shapefile = ogr.Open(filename)
layer = shapefile.GetLayer(0)
feature = layer.GetFeature(0)
geometry = feature.GetGeometryRef()

segments = getLineSegmentsFromGeometry(geometry)

print(segments)
```

Напомним о необходимости внести изменения в оператор `os.path.join()`**, чтобы он соответствовал месторасположению вашего файла фигур** `border.shp`.

Отметим, что рекурсивная функция `getLineSegmentsFromGeometry()` используется для того, чтобы из геометрии получить индивидуальные координаты каждого отрезка. Учитывая, что геометрии – это рекурсивные структуры данных, прежде чем мы сможем с ними работать, нам нужно извлечь отдельные отрезки.

В результате выполнения данной программы будет выведен длинный список точек, которые образуют различные отрезки, описывающие границу между этими двумя странами:

```
% python3 calcBorderLength.py
[[(100.081322, 20.348841), (100.089432, 20.347218)],
[(100.089432, 20.347218), (100.09137, 20.348606)],
[(98.742752, 10.348608), (98.748871, 10.378054)], ...]
```

Каждый отрезок состоит из списка точек – в этом случае вы заметите, что у каждого отрезка есть всего две точки, – и если вы приглядитесь, то увидите, что каждый отрезок начинается в той же самой точке, где заканчивается предыдущий. Всего имеется 459 отрезков, описывающих границу между Таиландом и Мьянмой, то есть 459 пар точек, для которых мы можем вычислить геодезическое расстояние.

Напомним, что геодезическое расстояние – это расстояние, измеряемое на поверхности Земли (с учетом ее выпуклости).

Посмотрим, каким образом можно воспользоваться библиотекой pyproj, чтобы вычислить геодезическое расстояние между любыми двумя точками. Сначала создадим экземпляр класса Geod:

```
geod = pyproj.Geod(ellps='WGS84')
```

Geod – это класс библиотеки pyproj, который выполняет геодезические расчеты. Отметим, что мы должны предоставить ему информацию о датуме, при помощи которого описывается фигура Земли. После задания экземпляра Geod можно вычислить геодезическое расстояние между любыми двумя точками, вызвав метод обратной геодезической трансформации geod.inv():

```
angle1,angle2,distance = geod.inv(long1, lat1, long2, lat2)
```

где angle1 – это угол от первой точки до второй, измеренный в десятичных градусах, angle2 – угол от второй точки до первой (снова в градусах) и distance – расстояние по дуге большого круга между двумя точками в метрах.

Применяя этот метод, мы сможем итеративно пройти по отрезкам, вычислить расстояние от одной точки до следующей и подытожить все расстояния, чтобы получить суммарную длину границы:

```
geod = pyproj.Geod(ellps='WGS84')

totLength = 0.0
for segment in segments:
    for i in range(len(segment)-1):
        pt1 = segment[i]
        pt2 = segment[i+1]

    long1,lat1 = pt1
    long2,lat2 = pt2

    angle1,angle2,distance = geod.inv(long1, lat1,
                                      long2, lat2)
    totLength += distance
```

После завершения работы переменная totLength будет содержать суммарную длину границы в метрах.

Собрав все вместе, мы получим полнофункциональную программу на Python, которая читает файл фигур border.shp, вычисляет и выводит на экран суммарную длину общей границы:

```
# calcBorderLength.py

import os.path
from osgeo import ogr
import pyproj

def getLineSegmentsFromGeometry(geometry):
```

```
    segments = []
    if geometry.GetPointCount() > 0:
        segment = []
        for i in range(geometry.GetPointCount()):
            segment.append(geometry.GetPoint_2D(i))
        segments.append(segment)
    for i in range(geometry.GetGeometryCount()):
        subGeometry = geometry.GetGeometryRef(i)
        segments.extend(
            getLineSegmentsFromGeometry(subGeometry))
    return segments

filename = os.path.join("common-border", "border.shp")
shapefile = ogr.Open(filename)
layer = shapefile.GetLayer(0)
feature = layer.GetFeature(0)
geometry = feature.GetGeometryRef()
segments = getLineSegmentsFromGeometry(geometry)

geod = pyproj.Geod(ellps='WGS84')

totLength = 0.0
for segment in segments:
    for i in range(len(segment)-1):
        pt1 = segment[i]
        pt2 = segment[i+1]

        long1,lat1 = pt1
        long2,lat2 = pt2

        angle1,angle2,distance = geod.inv(long1, lat1,
                                          long2, lat2)
        totLength += distance

print("Общая длина границы: {:.2f} км".format(totLength/1000))
```

В результате выполнения этой программы мы получим общую расчетную длину границы между Тайландом и Мьянмой:

```
% python3 calcBorderLength.py
Общая длина границы: 1730.55 км
```

Чтобы выполнить этот сценарий командной строки в Spyder, нужно настроить режим его запуска. В главном меню Spyder выберите **Запуск > Настроить**. В открывшемся окне настроек выполнения скрипта в разделе **Консоль** нажмите на переключателе **Выполнить в новой отдельной консоли Python**. В разделе **Основные настройки** нужно отметить флажок **Опции командной строки** и ввести входной аргумент сценария – `border.shp`. Затем там же отметить флажок **Рабочий каталог** и нажать кнопку напротив, чтобы выбрать месторасположение сценария. Эти настройки будут действовать только для данного сценария.

В этой программе мы приняли допущение, что файл фигур находится в географических координатах, используя эллипсоид WGS84, и содержит всего один геообъект. Теперь расширим нашу программу так, чтобы она учитывала любую предоставленную картографическую проекцию и датум и при этом обрабатывала все геообъекты, а не только один. В результате программа станет более гибкой, и она сможет работать с любым произвольным файлом фигур, а не только с файлом фигур общей границы, который мы создали ранее.

Сначала займемся проекцией и датумом. Можно сделать так же, как мы поступили ранее в этой главе с файлами фигур LULC с описанием землепользования и растительного покрова и roads с дорожной сетью, и поменять проекцию и датум файла фигур перед его обработкой. Это сработает, но потребует создания временного файла фигур только для того, чтобы вычислить длину, что не совсем эффективно. Вместо этого воспользуемся библиотекой pyproj непосредственно, при необходимости перепроецируя файл фигур назад в географические координаты. Это можно сделать, запросив пространственную привязку файла фигур:

```
shapefile = ogr.Open(filename)
layer = shapefile.GetLayer(0)
spatialRef = layer.GetSpatialRef()
if spatialRef == None:
    print("В файле фигур пространственная привязка на основе WGS84 отсутствует.")
    spatialRef = osr.SpatialReference()
    spatialRef.SetWellKnownGeogCS('WGS84')
```

Получив пространственную привязку, мы сможем увидеть, спроецирована пространственная привязка или нет, и если это так, то воспользуемся библиотекой pyproj, чтобы вернуть прямоугольные координаты назад в значения широты и долготы, как показано ниже:

```
if spatialRef.IsProjected():
    # Перевести спроецированные координаты назад
    # в значения широты и долготы.
    srcProj = pyproj.Proj(spatialRef.ExportToProj4())
    dstProj = pyproj.Proj(proj='longlat', ellps='WGS84',
                          datum='WGS84')
    ...
    long,lat = pyproj.transform(srcProj, dstProj, x, y)
```

Теперь можно переписать программу с учетом этого фрагмента, чтобы принимать данные, используя любую проекцию и датум. Одновременно с этим мы изменим ее так, чтобы вычислять общую длину каждого геообъекта в файле, а не только первого, а также принимать имя файла фигур из командной строки. Наконец, мы добавим в нее элементарный контроль ошибок. Назовем нашу новую программу calcFeatureLengths.py.

Напомним, что полную версию этой программы можно скачать в составе других примеров, прилагаемых к данной главе.

Начнем с того, что скопируем функцию `getLineSegmentsFromGeometry()`, которую мы использовали ранее:

```
import sys
from osgeo import ogr, osr
import pyproj

def getLineSegmentsFromGeometry(geometry):
    segments = []
    if geometry.GetPointCount() > 0:
        segment = []
        for i in range(geometry.GetPointCount()):
            segment.append(geometry.GetPoint_2D(i))
        segments.append(segment)
    for i in range(geometry.GetGeometryCount()):
        subGeometry = geometry.GetGeometryRef(i)
        segments.extend(
            getLineSegmentsFromGeometry(subGeometry))
    return segments
```

Затем получим имя открываемого из командной строки файла фигур:

```
if len(sys.argv) != 2:
    print("Применение: calcFeatureLengths.py <файл_фигур>")
    sys.exit(1)

filename = sys.argv[1]
```

Потом откроем файл фигур и получим его пространственную привязку, используя фрагмент программного кода, который мы написали ранее:

```
shapefile = ogr.Open(filename)
layer = shapefile.GetLayer(0)
spatialRef = layer.GetSpatialRef()
if spatialRef == None:
    print("В файле фигур пространственная привязка на основе WGS84 отсутствует.")
    spatialRef = osr.SpatialReference()
    spatialRef.SetWellKnownGeogCS('WGS84')
```

Затем получим исходную и целевую проекции, снова воспользовавшись программным кодом, который мы написали ранее. Отметим, что мы должны это сделать, только если используем прямоугольные координаты:

```
if spatialRef.IsProjected():
    srcProj = pyproj.Proj(spatialRef.ExportToProj4())
    dstProj = pyproj.Proj(proj='longlat', ellps='WGS84',
                          datum='WGS84')
```

Теперь все готово, чтобы начать обработку геообъектов файла фигур:

```
for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
```

Располагая геообъектом, теперь мы можем позаимствовать программный код, который мы использовали ранее для вычисления общей длины отрезков этого геообъекта:

```
geometry = feature.GetGeometryRef()
segments = getLineSegmentsFromGeometry(geometry)

geod = pyproj.Geod(ellps='WGS84')

totLength = 0.0
for segment in segments:
    for j in range(len(segment)-1):
        pt1 = segment[j]
        pt2 = segment[j+1]

        long1,lat1 = pt1
        long2,lat2 = pt2
```

Единственная разница в том, что если мы используем картографическую (прямоугольную) систему координат, то нам нужно перевести координаты назад в координаты WGS84:

```
    if spatialRef.IsProjected():
        long1,lat1 = pyproj.transform(srcProj,
                                      dstProj,
                                      long1, lat1)
        long2,lat2 = pyproj.transform(srcProj,
                                      dstProj,
                                      long2, lat2)
```

Затем можно воспользоваться библиотекой `pyproj`, чтобы вычислить расстояние между двумя точками, как мы сделали в нашем более раннем примере. Однако на этот раз мы завернем его в оператор `try...except`, с тем чтобы какой-либо сбой при вычислении расстояния не разрушил программу:

```
        try:
            angle1,angle2,distance = geod.inv(long1, lat1,
                                              long2, lat2)
        except ValueError:
            print("Невозможно вычислить расстояние из "
                  + "{:.4f},{:.4f} в {:.4f},{:.4f}"
                  .format(long1, lat1, long2, lat2))
            distance = 0.0

        totLength += distance
```

Вызов метода `geod.inv()` может вызвать ошибку `ValueError`[1], если две координаты находятся в том месте, где угол не может быть вычислен, например если две точки расположены на полюсах.

---

[1] Ошибка ValueError в Python означает, что функция получает аргумент правильного типа, но некорректного значения. – *Прим. перев.*

И наконец, теперь можно распечатать общую длину геообъекта в километрах:

```
print("Общая длина геообъекта {} равна {:.2f} км"
      .format(i, totLength/1000))
```

Эта программа может выполняться на любом файле фигур независимо от проекции и датума. Например, ее можно использовать, чтобы вычислить длину границы для каждой страны в мире, выполнив ее на наборе данных границ стран мира:

```
% python3 calcFeatureLengths.py TM_WORLD_BORDERS-0.3.shp
Общая длина геообъекта 0 равна 127.28 км
Общая длина геообъекта 1 равна 7264.69 км
Общая длина геообъекта 2 равна 2514.76 км
Общая длина геообъекта 3 равна 968.86 км
Общая длина геообъекта 4 равна 1158.92 км
Общая длина геообъекта 5 равна 6549.53 км
...
```

Эта программа является хорошим примером перевода координат геометрий (точек, ломаных, многоугольников) в расстояния. Теперь рассмотрим обратную операцию: использование расстояний для вычисления координат новых геометрий.

Задача: нахождение точки в 132.7 км к западу от г. Шошоун, шт. Калифорния

Используя скачанный ранее файл `CA_Features_xxxx.txt`, можно найти широту и долготу небольшого городка Шошоун в шт. Калифорнии к западу от Лас-Вегаса:

```
f = open("CA_Features_XXXX.txt", "r")
for line in f.readlines():
    chunks = line.rstrip().split("|")
    if chunks[1] == "Shoshone" and \
        chunks[2] == "Populated Place" and \
        chunks[3] == "CA":
        latitude = float(chunks[9])
        longitude = float(chunks[10])
```

При наличии этой координаты мы можем воспользоваться библиотекой `pyproj`, чтобы вычислить координату точки, удаленной на заданное расстояние и под заданным углом:

```
geod = pyproj.Geod(ellps="WGS84")
newLong,newLat,invAngle = geod.fwd(latitude, longitude,
                                   angle, distance)
```

Для решения этой задачи нам дано требуемое расстояние, и мы знаем, что целевая точка будет западнее от г. Шошоун. Библиотека `pyproj` использует азимутальные углы, которые измеряются по часовой стрелке от севера. Таким образом, «западнее» соответствует углу 270°.

Собрав все вместе, мы можем вычислить координаты нужной точки:

```
# findShoshone.py

import pyproj

distance = 132.7 * 1000
angle = 270.0

f = open("CA_Features_XXX.txt", "r")
for line in f.readlines():
    chunks = line.rstrip().split("|")
    if chunks[1] == "Shoshone" and \
        chunks[2] == "Populated Place" and \
        chunks[3] == "CA":
        latitude = float(chunks[9])
        longitude = float(chunks[10])
        geod = pyproj.Geod(ellps='WGS84')
        newLong,newLat,invAngle = geod.fwd(longitude,
                                           latitude,
                                           angle, distance)
        print("Координаты г. Шошоун: {:.4f},{:.4f}"
              .format(latitude, longitude))

        print("Координаты точки в {:.2f} км. к западу от г. Шошоун".format(distance/1000.0) +
              ": {:.4f}, {:.4f}".format(newLat, newLong))

f.close()
```

В результате выполнения этой программы получим ответ, который мы хотим:

```
% python3 findShoshone.py
Координаты г. Шошоун: 35.9730,-116.2711
Координаты точки в 132.70 км. к западу от г. Шошон: 35.9640, -117.7423
```

Упражнения

Если вы интересуетесь дальнейшим изучением методики, которая была использована в этой главе, то вы, возможно, захотите попробовать решить следующие ниже задачи:

- Внести изменение в расчет ограничительной рамки, чтобы исключить отдаленные острова.

Подсказка. Можно разделить составной многоугольник (объект MultiPolygon) каждой страны на отдельные многоугольники (объекты Polygon) и затем проверить площадь каждого многоугольника, чтобы исключить те, которые меньше заданного итогового значения.

- Применить набор данных границ стран мира для создания нового файла фигур, в котором все страны описываются одной геометрией точки, содержащей географический центр каждой страны.

Подсказка. Можно начать с ограничительных рамок стран, которые мы вычислили ранее, и затем вычислить срединную точку, используя следующую формулу:

```
midLat = (minLat + maxLat) / 2
midLong = (minLong + maxLong) / 2
```

Можно усложнить задачу и воспользоваться методом `centroid()` библиотеки `Shapely`, чтобы вычислить более точное значение центра каждой страны. Для этого необходимо конвертировать контур страны в геометрию библиотеки `Shapely`, вычислить центроид и затем перед сохранением центроида в выходном файле фигур конвертировать его назад в геометрию библиотеки `OGR`.

- Расширить пример гистограммы, чтобы включать только те значения высоты, которые попадают в контур выбранной страны.

Подсказка. Эффективная реализация этой задачи может оказаться затруднительной. Хороший подход мог бы заключаться в определении ограничительной рамки для всех многоугольников, которые образуют контур страны, и выполнении итерации по координатам ЦКМ внутри этой ограничительной рамки. Затем можно было бы проверить, не лежит ли заданная координата внутри контура страны, используя для этого логическую функцию `polygon.contains(point)`, и добавлять высоту к гистограмме, только если точка фактически лежит внутри контура страны.

- Оптимизировать пример с идентификацией близлежащих национальных парков таким образом, чтобы он мог работать быстро с более крупными наборами данных.

Подсказка. Одна из возможностей может состоять в том, чтобы вычислить прямоугольную ограничительную рамку вокруг каждой городской агломерации и затем расширить эту ограничительную рамку на север, юг, восток и запад на нужное угловое расстояние. И затем, прежде чем вызывать очень медленную логическую функцию `polygon.contains(point)`, можно сначала быстро исключить все точки, которые не находятся в этой ограничительной рамке.
Еще один метод оптимизации данной программы – исключить городские агломерации за пределами шт. Калифорния. Для этого необходимо найти файл фигур для штатов США, загрузить контур шт. Калифорния и включать только те городские агломерации, которые лежат (полностью или частично) в пределах границ штата.

- Вычислить суммарную длину береговой линии Соединенного Королевства Великобритании и Северной Ирландии.

Подсказка. Напомним, что контур страны – это составной многоугольник, где каждый находящийся в нем многоугольник представляет один остров. Необходимо извлечь внешнее кольцо из каждого из этих отдельных островных многоугольников и вычислить суммарную длину (протяженность) отрезков в этом внешнем кольце. Затем можно подытожить длину каждого отдельного острова, чтобы в итоге получить длину береговой линии всей страны.

- Разработать свою собственную библиотеку геопространственных функций многократного использования на основе библиотек OGR, GDAL, Shapely и pyproj, выполняющую общие операции, как те, которые обсуждались в этой главе.

Подсказка. Написание своих собственных библиотечных модулей многократного использования является распространенной тактикой программирования. Подумайте о различных задачах, которые мы решали в этой главе, и каким образом их можно превратить в универсальные библиотечные функции. Например, вы, возможно, захотите написать функцию `calcLineStringLength()`, которая принимает ломаную и возвращает общую длину ее отрезков, дополнительно переводя прямоугольные координаты ломаной в значения широты и долготы перед вызовом метода, обратного геодезической трансформации `geod.inv()`.
Затем можно написать функцию `calcPolygonOutlineLength()`, которая использует предыдущую функцию `calcLineStringLength()`, чтобы вычислить длину внешнего кольца многоугольника.

Заключение

В этой главе мы рассмотрели разнородную методику применения библиотек OGR, GDAL, Shapely и pyproj внутри программ на Python для решения целого ряда практических задач. При этом мы научились вычислять ограничительную рамку для страны, использовать библиотеку Shapely для вычисления общей границы между двумя странами, анализировать содержимое файла ЦКМ, менять проекции и датумы, отыскивать при помощи функции `buffer()` точки, лежащие рядом с многоугольником, использовать библиотеку pyproj для вычисления длины геометрии и вычислять целевую точку при наличии расстояния и направления движения (bearing) от стартовой точки.

До сих пор мы писали программы, которые работают непосредственно с файлами фигур и другими источниками данных, чтобы загружать и затем обрабатывать геоданные. В следующей главе мы рассмотрим приемы, в которых при помощи баз данных можно «разгонять» разработку своего геоприложения. Вместо того чтобы последовательно считывать пространственные данные для каждого геообъекта в память, можно выполнять пространственные запросы непосредственно в базе данных, которая позволяет вам хранить и работать с намного бóльшим количеством пространственных данных, чем это можно было сделать при помощи методики, которую мы рассматривали до сих пор.

Глава 6

Пространственные базы данных

В этой главе мы рассмотрим приемы использования геопространственного расширения PostGIS объектно-реляционной СУБД PostgreSQL для хранения пространственных данных и работы с ними. В частности, мы коснемся таких аспектов, как:

- концепция СУБД с поддержкой пространственных данных;
- пространственные индексы и как они работают;
- каким образом `PostGIS` действует как расширение реляционной СУБД PostgreSQL;
- как установить на свой компьютер СУБД PostgreSQL, расширение PostGIS и адаптер СУБД `psycopg2` для Python;
- как при помощи PostGIS установить и сконфигурировать пространственную базу данных;
- как использовать адаптер СУБД `psycopg2` для доступа к пространственной базе данных из своей программы на Python;
- как создавать, импортировать и выполнять запросы к пространственным данным при помощи Python;
- рекомендуемые приемы наиболее успешной практики хранения пространственных данных в базе данных.

Настоящая глава должна послужить введением в использование баз данных в геоприложении. Основанное на этом материале изложение *главы 8 «Работа с пространственными данными»* позволит выполнять мощные пространственные запросы, которые в обычных условиях невозможны при работе с файлами фигур и другими файлами геоданных.

СУБД с поддержкой пространственных данных

В некотором смысле почти любая база данных может использоваться для хранения геоданных: просто конвертируйте геометрию в формат WKT и сохраните результат в столбце `text`. Несмотря на то что этот прием позволит вам хранить в базе данных геоданные, он не позволит вам каким-либо полезным образом выполнять

к ней запросы. Вы лишь будете последовательно, запись за записью, извлекать исходный текст WKT и конвертировать его обратно в объект геометрии.

СУБД с поддержкой пространственных данных, с другой стороны, учитывает концепцию *пространства* и позволяет вам работать непосредственно с пространственными объектами и понятиями. В частности, СУБД с поддержкой пространственных данных дает вам возможность:

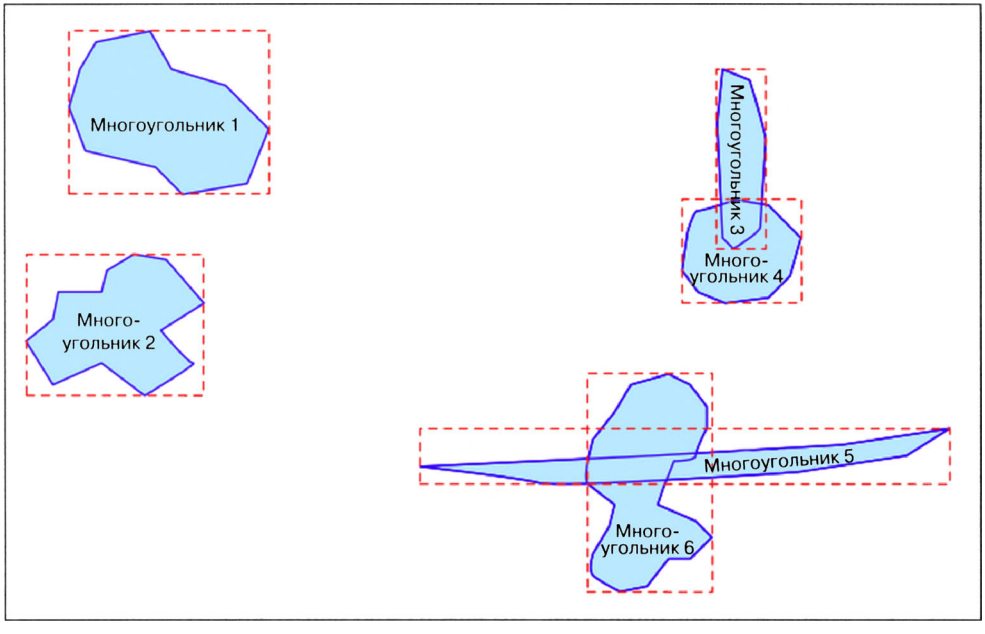
- **хранить пространственные типы данных** (точки, линии, многоугольники и т. д.) непосредственно в базе данных в столбце `GEOMETRY`;
- выполнять к своим данным **пространственные запросы**, например «выбрать все ориентиры в пределах 10 км от г. "Сан-Франциско»;
- выполнять на своих данных **операции пространственного соединения** (join), например выбрать все города и соответствующие страны путем соединения города и страны по предикату "город находится в стране";
- создавать новые пространственные объекты, используя различные **пространственные функции**, например назначить переменной `"danger_zone"` (опасная зона) результат пересечения многоугольников `"flooded_area"` (зона наводнения) и `"urban_area"` (городская агломерация).

Как вы можете догадаться, СУБД с поддержкой пространственных данных представляет собой чрезвычайно мощный инструмент для работы с геоданными. При помощи **пространственных индексов** и другой оптимизации пространственные СУБД могут выполнять эти типы операций очень быстро и масштабироваться для поддержания огромного объема данных, что просто не выполнимо при использовании других схем хранения данных.

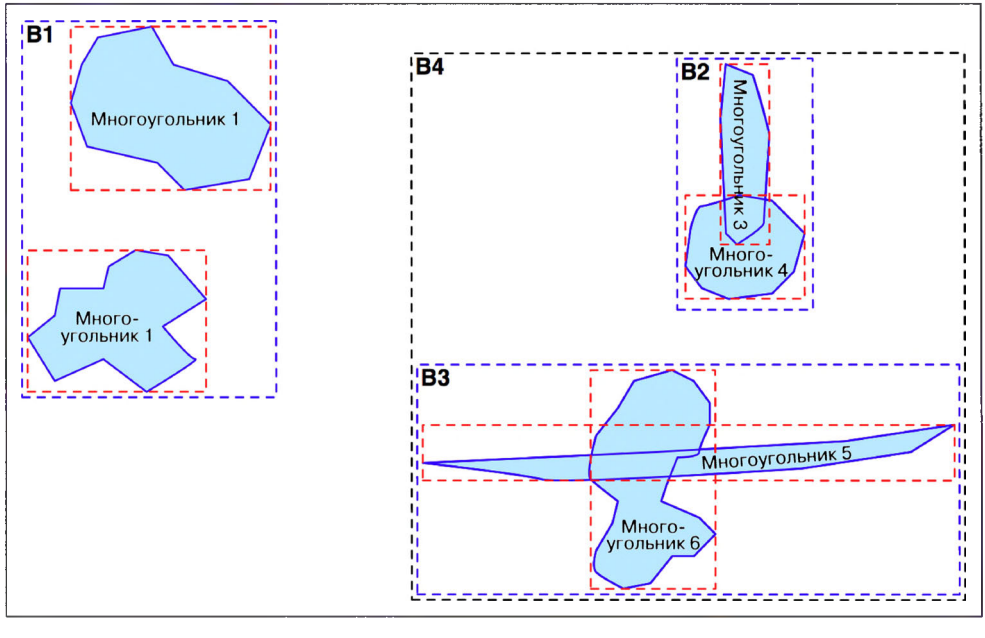
Пространственные индексы

Одной из определяющих особенностей пространственной СУБД является способность создавать и использовать «пространственные» индексы с целью ускорения поиска на основе геометрии. Эти индексы используются для выполнения пространственных операций, таких как идентификация всех объектов, которые лежат внутри конкретной ограничительной рамки, идентификация всех объектов на определенном расстоянии от конкретной точки или идентификация всех объектов, которые пересекаются с конкретным многоугольником.

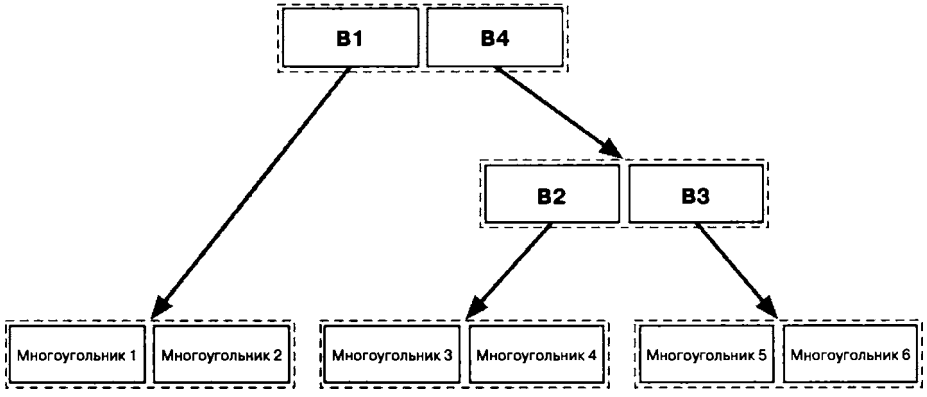
Пространственные индексы – это один из мощнейших функциональных элементов пространственных СУБД, и для ознакомления с тем, как они работают, стоит потратить некоторое время. Пространственные индексы непосредственно не хранят геометрию; вместо этого по каждой геометрии они вычисляют **ограничительную рамку** и затем индексируют геометрии на основе их ограничительных рамок. Это позволяет СУБД выполнять быстрый поиск по геометриям на основе их положения в пространстве:



Ограничительные рамки группируются во вложенную иерархию на основе того, как близко друг к другу они находятся, как показано на следующей ниже иллюстрации:

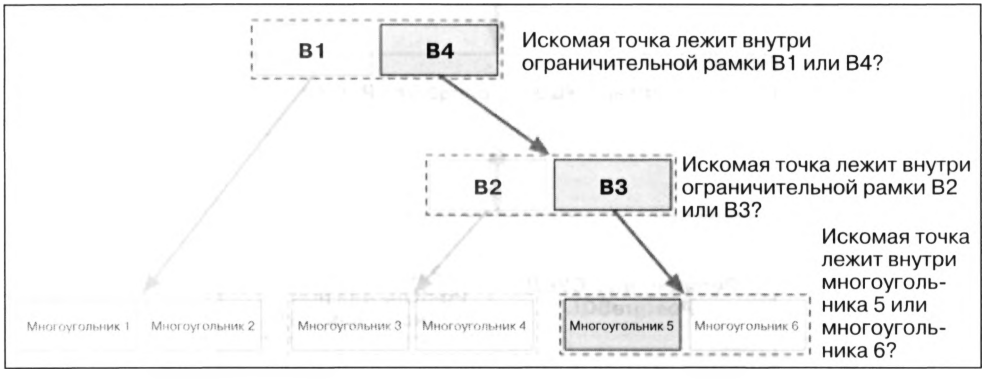


Иерархия вложенных ограничительных рамок затем оформляется в виде древовидной структуры данных, как показано ниже:



Компьютер может оперативно просмотреть это дерево в поисках конкретной геометрии или же сравнить географические положения или размеры различных геометрий. Например, предположим, что нам нужно найти многоугольник, который пересекает следующую ниже точку:

Геометрию с точкой *X* можно быстро отыскать путем обхода дерева и сравнения ограничительных рамок на каждом уровне. Поиск по пространственному индексу будет выполняться следующим образом:



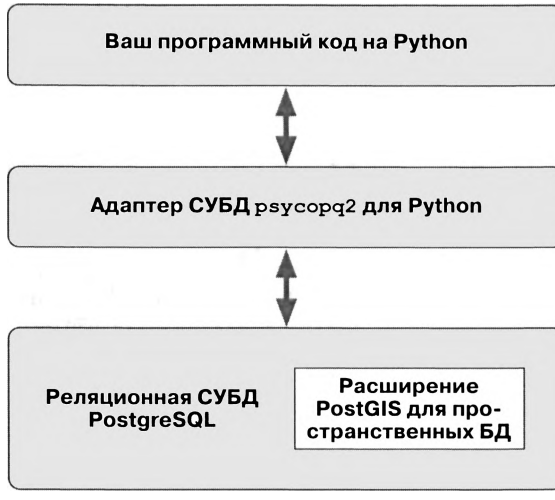
Чтобы идентифицировать искомый многоугольник, используя пространственный индекс, потребуются всего три сравнения.

По своей иерархической природе пространственные индексы чрезвычайно хорошо масштабируются и могут выполнять поиск по многим десяткам тысяч объектов, используя всего несколько сравнений ограничительных рамок. И, поскольку каждая геометрия сводится к простой ограничительной рамке, пространственные индексы могут поддерживать любой тип геометрии, а не только многоугольники.

Пространственные индексы не ограничиваются только поиском вложенных координат; они могут использоваться для всех видов пространственных сравнений и пространственных соединений. Мы будем активно работать с пространственными индексами на протяжении оставшейся части книги.

Знакомство с PostGIS

В этой книге мы будем работать с PostGIS. Это одна из самых популярных и мощных геопространственных СУБД, которая фактически представляет собой расширение объектно-реляционной системы управления базами данных PostgreSQL. Ее дополнительное преимущество заключается в том, что она имеет открытый исходный код и находится в свободном доступе. Чтобы использовать PostGIS из своих программ на Python, сначала необходимо инсталлировать и настроить СУБД PostgreSQL, затем инсталлировать геопространственное расширение PostGIS и, наконец, установить адаптер СУБД PostgreSQL `psycopg2` для Python. Следующая иллюстрация показывает, как все эти части компонуются вместе:



Отметим, что СУБД PostgreSQL нередко именуется **Postgres**. Мы регулярно будем пользоваться этим неформальным наименованием в оставшейся части книги.

Расширение PostGIS позволяет хранить различные типы пространственных данных и выполнять к ним запросы. Эти данные в том числе состоят из точек, линий, многоугольников и коллекций геометрий. В PostGIS пространственное поле, служащее для хранения пространственных данных, может иметь один из двух типов:

- поле с типом **GEOMETRY** содержит пространственные данные, которые, как предполагается, находятся в картографической (спроецированной) системе координат. Все расчеты на полях `GEOMETRY` и запросы к ним предполагают, что пространственные данные спроецированы на координатную плоскость. Это намного упрощает расчеты, но при этом будет работать надежно, только если пространственные данные находятся в картографической системе координат;
- поле с типом **GEOGRAPHY** содержит пространственные данные, которые используют геодезические (неспроецированные) координаты. Расчеты на полях `GEOGRAPHY` и запросы к ним предполагают, что данные находятся в угловых единицах (то есть имеют значения широты и долготы), используя изощренный математический аппарат для расчета длин и площадей на основе сфероидной модели Земли.

См. сценарий `transform_to_latlong.py`, в котором на примере координат г. Москва выполняется перевод географических координат (широты и долготы) точки в прямоугольные координаты проекции Гаусса-Крюгера, дает представление об указанном математическом аппарате (пример взят из вики-книги – https://ru.wikibooks.org/wiki/Реализации_алгоритмов/Перевод_географических_координат_в_прямоугольные).

Учитывая, что привлекаемый математический аппарат здесь намного сложнее, для полей GEOGRAPHY доступны не все пространственные функции, и выполнение операций зачастую требует большего количества времени. Однако поля GEOGRAPHY намного легче использовать, если в ваших пространственных данных используется географическая (неспроецированная) система координат, такая как WGS84.

Пойдем дальше и выполним инсталляцию расширения PostGIS на ваш компьютер, а затем рассмотрим, каким образом можно использовать это расширение для создания пространственной базы данных и работы с ней с использованием языка Python.

Инсталляция СУБД PostgreSQL

PostgreSQL – это чрезвычайно мощная объектно-реляционная система управления базами данных (СУБД) с открытым исходным кодом.

Главный веб-сайт СУБД Postgres находится на http://postgresql.org. Способ инсталляции СУБД Postgres зависит от операционной системы, в которой работает ваш компьютер:

- в Linux следуйте инструкциям по инсталляции Postgres на свой компьютер на странице для скачивания PostgreSQL (http://postgresql.org/download). Выберите ссылку, соответствующую своему дистрибутиву Linux, и вы получите необходимые инструкции по инсталляции;
- в Mac OS X можно загрузить установщик Postgres с веб-сайта KyngChaos (http://www.kyngchaos.com/software/postgres). Удостоверьтесь, что вы не скачиваете версию только для клиента, поскольку тогда вам понадобится сервер Postgres. После того как вы скачали установщик, откройте образ диска и сделайте двойной щелчок по файлу пакета PostgreSQL.pkg, чтобы установить Postgres на свой компьютер;
- установщик Postgres для Windows можно скачать с http://enterprisedb.com/products-services-training/pgdownload. Выберите установщик, соответствующий своей версии Windows (32- или 64-разрядной), скачайте файл установщика, затем просто дважды щелкните по установщику и дальше следуйте инструкциям.

После завершения инсталляции СУБД Postgres можно проверить ее работоспособность, набрав psql в окне терминала или в командном окне и нажав клавишу *Return*. Если все выполнено правильно, то вы должны увидеть командную строку СУБД Postgres[1]:

```
psql (9.5.4)
Введите "help", чтобы получить справку.
postgres=#
```

Если операционная система не может найти клиента командной строки psql, то вам, скорее всего, придется добавить в переменную Path среды пользователя

[1] Документацию по СУБД Postgres версии 9.5 на русском языке можно скачать с веб-сайта компании Postgres Professional (https://postgrespro.ru/docs). – *Прим. перев.*

путь к каталогу `bin` СУБД PostgreSQL. Например, в Mac можно открыть файл `bash_profile` в текстовом редакторе и добавить следующее: `export PATH="$PATH:/usr/local/pgsql/bin"`. В Windows следует в переменную `PATH` среды пользователя Windows добавить путь `C:\Program Files\PostgreSQL\9.5\bin`.

Если команда `psql` жалуется на аутентификацию пользователя, то вам, по-видимому, нужно указать учетную запись пользователя, которая будет применяться при подключении к СУБД Postgres. Например:

```
% psql -U postgres
```

Во многих инсталляциях Postgres при получении доступа к этой СУБД требуется ввести стандартное имя пользователя `postgres` с опцией командной строки `-U`. Как вариант вам, возможно, придется воспользоваться утилитой `sudo`[1], чтобы выполнить `psql` от имени суперпользователя или открыть командную строку от имени администратора, в случае если вы работаете в Windows.

Чтобы выйти из клиента командной строки Postgres, наберите `\q` и нажмите клавишу *Return*.

Инсталляция расширения PostGIS

Наша следующая задача состоит в том, чтобы установить геопространственное расширение PostGIS для СУБД Postgres. Главный веб-сайт PostGIS находится на http://postgis.net. Опять же, способ инсталляции расширения PostGIS зависит от операционной системы, в которой вы работаете:

- для компьютеров на основе Linux следуйте инструкциям на странице установки PostGIS (http://postgis.net/install);

В Fedora 24 установить СУБД PostgreSQL, дополнительные утилиты и функционал, а также расширение PostGIS можно одной командой:

```
dnf install postgresql postgresql-contrib postgresql-server postgis
```

- для Mac OS X необходимо загрузить и выполнить установщик PostGIS с веб-сайта KyngChaos (http://kyngchaos.com/software/postgres);

Отметим, что этот установщик PostGIS требует полнофункциональной версии пакета **GDAL Complete**, которая должна быть уже установлена во время работы с *главой 2 «Геоинформационные системы»*.

- для Windows можно загрузить установщик со страницы http://download.osgeo.org/postgis/windows.

В конце инсталляции СУБД PostgreSQL в Windows будет предложено запустить мастер-компоновщик **Application Stack Builder** с перечнем дополнительного программного обеспечения, включая пространственное расширение

[1] Команда sudo (англ. substitute user and do, дословно «подменить пользователя и выполнить») – это утилита, предоставляющая привилегии root для выполнения административных операций в соответствии со своими настройками. – *Прим. перев.*

PostGIS. В приветственном окне нужно вызвать ниспадающий список, выбрать из него свою версию PostgreSQL и нажать кнопку *Следующий*. Из появившего в следующем окне перечня категорий ПО выбрать **Spatial Extensions** (Пространственные расширения), выбрать версию PostGIS согласно разрядности ОС и нажать кнопку *Следующий*. Далее следуйте инструкциям по установке. Кроме того, мастер-компоновщик **Application Stack Builder** можно запустить позже, после установки. Программа расположена в подпапке `C:\Program Files\PostgreSQL\9.5\bin`.

Чтобы проверить, что расширение PostGIS было успешно проинсталлировано, попробуйте набрать в окне командной строки Windows `psql -U postgres` или в окне терминала в Linux `sudo -i -u postgres`, а затем следующие ниже команды:

```
% createdb test_database
% psql -d test_database -c "CREATE EXTENSION postgis;"
% dropdb test_database
```

Напоминаем, что опция `-U postgres` и использование утилиты `sudo` требуются, в случае если вы будете работать с Postgres под другой учетной записью пользователя.

Первая команда создает новую (пробную) базу данных, вторая активирует для нее расширение PostGIS, и третья команда ее удаляет. Если эта последовательность команд выполняется без каких-либо ошибок, то инсталлированное расширение PostGIS (включая саму СУБД Postgres) настроено и работает правильно.

Установка адаптера psycopg2

Адаптер СУБД Postgres для Python `psycopg2` представляет собой библиотеку Python, которая используется для доступа к Postgres из программ на Python. Главный веб-сайт `psycopg2` находится на http://initd.org/psycopg.

Как обычно, способ установки адаптера `psycopg2` зависит от операционной системы, в которой вы работаете:

- в Fedora 24 необходимо установить адаптер `psycopg2` из исходных текстов. По поводу инструкций о том, как это сделать, обратитесь к странице http://initd.org/psycopg/docs/install.html:

  ```
  dnf install python3-psycopg2
  sudo pip3 install psycopg2 --upgrade
  ```

- для компьютеров под Mac OS X для установки адаптера `psycopg2` из командной строки можно воспользоваться диспетчером пакетов Python `pip`:

  ```
  pip install psycopg2
  ```

  Отметим, что, для того чтобы `psycopg2` мог скомпилироваться, у вас должны быть установлены инструменты командной строки XCode;
- в Windows запускаемый по двойному щелчку установщик для адаптера `psycopg2` можно скачать со страницы http://www.stickpeople.com/projects/python/win-psycopg.

Чтобы проверить, что установка прошла успешно, запустите свой интерпретатор Python и наберите следующую команду:

```
>>> import psycopg2
>>>
```

Если адаптер psycopg2 был установлен правильно, то вы снова должны увидеть приглашение интерпретатора Python без каких-либо сообщений об ошибках, как показано в примере выше. Если сообщение об ошибке все же появилось, то вам, скорее всего, придется обратиться к инструкции по поиску неисправностей, которые имеются на веб-сайте адаптера psycopg2.

Настройка СУБД

Установив необходимое программное обеспечение, теперь посмотрим, каким образом можно применить расширение PostGIS для создания и настройки пространственной базы данных. Начнем с создания учетной записи пользователя Postgres, создания самой базы данных и настройки доступа к ней пользователя, после чего мы активируем для нее геопространственное расширение PostGIS.

Создание учетной записи пользователя Postgres

Наша первая задача состоит в том, чтобы выполнить настройку пользователя Postgres, который будет владеть создаваемой нами базой данных. Хотя у вас может иметься учетная запись пользователя на своем компьютере, которую вы используете для входа и выхода из операционной системы, учетная запись пользователя PostgreSQL совершенно отдельная от этой учетной записи и используется только в Postgres. По желанию можно настроить пользователя PostgreSQL с тем же самым именем, что и имя входа в операционную систему компьютера, либо дать ему другое имя.

Отметим, что в руководстве по СУБД Postgres для пользователя иногда используется термин «роль».

Чтобы создать нового пользователя СУБД PostgreSQL, наберите следующую команду SQL:

```
% createuser -P <имя пользователя>
```

Разумеется, надо заменить `<имя пользователя>` на любое имя, которое вы хотите использовать для вашего нового пользователя. Вам также, возможно, потребуется добавить опцию `-U postgres` либо для этих команд воспользоваться утилитой `sudo`, в случае если необходимо запустить Postgres с другой учетной записью пользователя.

Параметр командной строки `-P` сообщает Postgres, что для этого нового пользователя вы хотите ввести пароль. Вводимый пароль следует запомнить, так как он понадобится, когда вы попытаетесь получить доступ к своей базе данных.

Создание базы данных

Затем необходимо создать базу данных, которую вы хотите использовать для хранения своих пространственных данных. Это делается при помощи команды SQL createdb:

```
% createdb <имя БД>
```

Удостоверьтесь, что вы поменяли <имя БД> на имя создаваемой вами базы данных. Опять же, при необходимости следует добавить опцию -U Postgres либо воспользоваться утилитой sudo.

Разрешение доступа к базе данных

Чтобы разрешить пользователю получать доступ к новой базе данных, запустите интерфейс командной строки СУБД Postgres и в окне терминала наберите следующую ниже команду, при необходимости добавив -U postgres (нижеследующая команда означает предоставить все привилегии на базу данных с таким-то именем такому-то пользователю):

```
% psql -c "GRANT ALL PRIVILEGES ON DATABASE <имя БД> TO <пользователь>;"
```

Включить поддержку пространственных данных

На данный момент мы создали базу данных Postgres и связанного с ней пользователя. Наша база данных – это просто обычная база данных; чтобы превратить ее в пространственную базу данных, мы должны включить для нее пространственное расширение PostGIS. Для этого наберите в окне терминала следующую ниже команду, поменяв <имя БД> на имя вашей новой базы данных:

```
% psql -d <имя БД> -c "CREATE EXTENSION postgis;"
```

Использование расширения PostGIS

Имея в распоряжении пространственную базу данных, теперь посмотрим, как получить к ней доступ из Python. При помощи адаптера psycopg2 получение доступа к пространственной базе данных из Python выполняется довольно прямолинейно. Например, следующий фрагмент программного кода показывает, как подключиться к базе данных и выдать простой запрос:

```
import psycopg2

connection = psycopg2.connect(database="...", user="...",
                              password="...")

cursor = connection.cursor()
cursor.execute("SELECT id,name FROM cities WHERE pop>100000")
for row in cursor:
    print(row[0],row[1])
```

Оператор psycopg2.connect() открывает соединение с базой данных, используя имя базы данных, имя пользователя и пароль, который вы настроили, когда создавали и конфигурировали базу данных. После установления соединения с базой данных вы создаете объект Cursor, в отношении которого вы можете выполнять запросы. И затем вы можете извлекать совпадающие данные, как показано в приведенном выше примере.

Воспользуемся адаптером psycopg2, чтобы сохранить набор данных границ стран мира в пространственную таблицу базы данных, и затем выполним несколько простых запросов в отношении этих данных. Поместите копию набора данных в подходящий каталог и в том же самом каталоге создайте новую программу Python с именем postgis_test.py. Наберите в своей программе следующий фрагмент:

```
import psycopg2
from osgeo import ogr

connection = psycopg2.connect(database="<имя БД>",
                              user="<пользователь>",
                              password="<пароль>")
cursor = connection.cursor()
```

Следует поменять значения <имя БД>, <пользователь> и <пароль> на определенные заранее имя базы данных, имя учетной записи пользователя и пароль.

Пока мы просто открыли соединение с базой данных. Теперь создадим таблицу, где будут храниться данные границ стран мира. Для этого в конец своей программы добавьте следующий ниже фрагмент:

```
cursor.execute("DROP TABLE IF EXISTS borders")
cursor.execute("CREATE TABLE borders (" +
               "id SERIAL PRIMARY KEY," +
               "name VARCHAR NOT NULL," +
               "iso_code VARCHAR NOT NULL," +
               "outline GEOGRAPHY)")
cursor.execute("CREATE INDEX border_index ON borders " +
               "USING GIST(outline)")
connection.commit()
```

Как видите, мы удаляем таблицу базы данных, если она уже существует, чтобы можно было безошибочно выполнять нашу программу повторно. Затем создаем новую таблицу borders (границы) с четырьмя полями: id, name и iso_code, все из которых являются стандартными полями базы данных, и пространственным полем outline (контур) с типом GEOGRAPHY. Поскольку это поле используется для данных с типом GEOGRAPHY, мы можем хранить в нем пространственные данные с географическими координатами широты и долготы.

Третий оператор создает пространственный индекс по полю outline. В PostGIS для задания пространственного индекса мы используем тип индекса GIST[1].

---

[1] Аббревиатура индекса GiST (Generalized Search Tree) переводится как *обобщенное дерево поиска*. См. документацию по типам индексов в СУБД Postgres на странице https://postgrespro.ru/docs/postgrespro/9.5/indexes-types. – *Прим. перев.*

И наконец, поскольку СУБД Postgres является транзакционной базой данных, мы должны зафиксировать вносимые изменения при помощи оператора `connection.commit()`.

Теперь, когда мы определили нашу таблицу базы данных, добавим в нее немного данных. При помощи методики, которую мы изучили ранее, прочитаем все содержимое файла фигур с набором данных границ стран мира. Вот соответствующий фрагмент кода:

```
shapefile = ogr.Open("TM_WORLD_BORDERS-0.3/TM_WORLD_BORDERS-0.3.shp")
layer = shapefile.GetLayer(0)

for i in range(layer.GetFeatureCount()):
    feature  = layer.GetFeature(i)
    name     = feature.GetField("NAME")
    iso_code = feature.GetField("ISO3")
    geometry = feature.GetGeometryRef()
    wkt      = geometry.ExportToWkt()
```

Все это должно быть совершенно понятным. Наша следующая задача состоит в том, чтобы сохранить эту информацию в базу данных. Для этого мы воспользуемся командой `INSERT`. Добавьте следующий фрагмент к своей программе внутри цикла `for`:

```
cursor.execute("INSERT INTO borders (name, iso_code, outline) " +
               "VALUES (%s, %s, ST_GeogFromText(%s))",
               (name, iso_code, wkt))
```

Отметим, что адаптер `psycopg2` автоматически конвертирует стандартные типы данных Python, такие как числа, строковые и значения даты-времени в соответствующий формат для вставки в базу данных. Следуя питоновскому стандарту DB-API[1], символ `%s` используется в качестве заполнителя, представляющего значение, а это значение берется из списка, переданного в качестве второго параметра функции `execute()`. Другими словами, первый символ `%s` заменяется значением переменной `name`, второй – значением переменной `iso_code` и т. д.

Поскольку адаптер `psycopg2` не знает о значениях данных геометрии, мы должны конвертировать геометрию в строковое значение в формате WKT и затем воспользоваться функцией `ST_GeogFromText()`, чтобы конвертировать эту строку назад в объект PostGIS с типом `GEOGRAPHY`.

Импортировав все данные, теперь нужно зафиксировать изменения, которые мы внесли в базу данных. Для этого добавьте следующий ниже оператор в конце своей программы (вне цикла `for`):

```
connection.commit()
```

---

[1] DB-API – общий программный интерфейс (API), скрывающий детали реализации каждой отдельной СУБД. Любой Python-пакет, реализующий это API, взаимозаменяем. См. документацию на странице http://lectureswww.readthedocs.io/6.www.sync/2.codding/9.databases/db-api.html. – *Прим. перев.*

На выполнение вышеприведенной программы импорта всех данных в базу данных потребуется приблизительно 30 секунд, правда, в итоге ничего не произойдет. Чтобы получить подтверждение, что все работает, выполним относительно импортированных данных простой пространственный запрос; к примеру, нам нужно найти все страны, которые расположены в пределах 500 км от г. Цюрих в Швейцарии. Начнем с того, что зададим широту и долготу Цюриха, а также целевой радиус поиска в метрах. Добавим в конец своей программы нижеследующее:

```
start_long = 8.542
start_lat  = 47.377
radius     = 500000
```

Теперь можно выполнить пространственный запрос, используя для этого функцию `ST_DWithin()`, как показано ниже:

```
cursor.execute("SELECT name FROM borders WHERE ST_DWithin(" +
               "ST_MakePoint(%s, %s), outline, %s)",
               (start_long, start_lat, radius))
for row in cursor:
    print(row[0])
```

Функция `ST_DWithin()` находит все записи в таблице `borders` с полем `outline` в пределах `radius` метров от заданного значения широты и долготы. Отметим, что мы используем функцию `ST_MakePoint()`, чтобы перевести значение широты и долготы в геометрию точки, что позволяет нам сопоставить контур с заданной точкой.

В результате выполнения этой программы все данные будут импортированы, и будет показан список стран, которые расположены в пределах 500 км от г. Цюрих:

```
Luxembourg
Monaco
San Marino
Austria
Czech Republic
France
Germany
Croatia
Italy
Liechtenstein
Belgium
Netherlands
Slovenia
Switzerland
```

Хотя предстоит проделать еще массу работы, эта программа должна продемонстрировать, как использовать расширение PostGIS для создания пространственной базы данных, вставки в нее данных и выполнения к этим данным запроса, и все это при помощи программного кода на Python.

Документация по PostGIS

Поскольку PostGIS является расширением СУБД PostgreSQL и для получения к ней доступа вы используете адаптер СУБД psycopg2 для Python, существуют три отдельных комплекта документации, к которой необходимо обратиться:

- руководство по СУБД PostgreSQL: http://postgresql.org/docs;
- руководство по PostGIS: http://postgis.refractions.net/docs;
- документация по адаптеру psycopg2: http://initd.org/psycopg/docs.

Из них руководство по гепространственному расширению PostGIS, вероятно, будет самым полезным. Помимо него, необходимо будет обратиться к документации по адаптеру psycopg2, чтобы узнать детали использования геопространственного расширения PostGIS из программ на Python. Впрочем, также стоит обратиться и к руководству по СУБД PostgreSQL, чтобы изучить непространственные аспекты использования PostGIS, хотя стоит помнить, что это руководство огромное и чрезвычайно сложное, что отражает сложность самой СУБД PostgreSQL.

Продвинутый функционал PostGIS

Будучи геопространственным расширением, PostGIS поддерживает целый ряд продвинутых возможностей, которые могут оказаться полезными:

- динамические («на лету») трансформации геометрий из одной пространственной привязки в другую;
- способность редактировать геометрии путем добавления, изменения и удаления точек и вращения, масштабирования и смещения всех геометрий;
- способность читать и писать геометрии в форматах GeoJSON, GML, KML и SVG наряду с форматами WKT и WKB;
- полный набор сопоставлений ограничительных рамок, в том числе операторы `A пересекается с B`, `A содержит B` и `A слева от B`. Эти операторы сравнения используют пространственные индексы для чрезвычайно быстрой идентификации совпадающих объектов;
- соответствующие операции пространственного сравнения геометрий, включая пересечение, включение, пересечение, равенство, наложение, касание и т. д. Эти операции сравнения выполняются на основе собственно данных геометрии, а не только при помощи ограничительных рамок;
- пространственные функции для расчета такой информации, как площадь, средняя точка, ближайшая точка, расстояние, длина, периметр, самая короткая соединительная линия и т. д. Эти функции учитывают пространственную привязку геометрии, в случае если она известна.

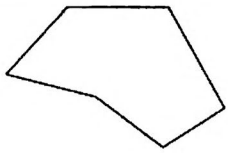
Геопространственное расширение PostGIS обладает заслуженной репутацией геопространственного «локомотива». Не являясь единственной общедоступной пространственной СУБД, она остается единственным в своем роде самым мощным и полезным инструментом, который мы будем активно использовать в оставшейся части книги.

Наиболее успешные практические приемы

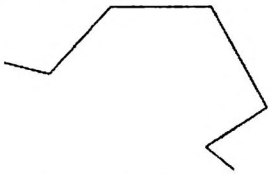
В этом разделе мы рассмотрим ряд практических приемов, которые можно взять на вооружение, чтобы обеспечить максимально эффективную и продуктивную работу геопространственных баз данных.

Рекомендуем: используйте базу данных для отслеживания пространственных привязок

Как мы убедились в более ранних главах, в разных наборах геоданных используются разные системы координат, датумы и проекции. Рассмотрим, например, два следующих объекта с типом `GEOMETRY`:

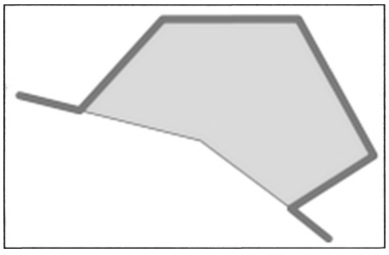


Многоугольник с координатами
(13.1425, –113.5236), (13.9220; –114.3740)
(15.1976, –114.3740), (15.9063, – 113.0984),
(15.1268, –112.6024), (14.2764, –113.2402)



Ломаная с координатами
(197443, 1420617), (226414, 1454264)
(266249, 1252373), (331434, 1252373),
(367647, 1555211), (327812, 1672979)
(345918, 1740279)

Их геометрии представлены серией координат, которые суть просто числа. Сами по себе эти числа бесполезны – необходимо поместить эти координаты на поверхность Земли, определив используемую в геометрии **пространственную привязку** (систему координат, датум и проекцию). В данном случае в многоугольнике использованы географические координаты широты и долготы по датуму WGS84, в то время как в ломаной использованы координаты UTM в зоне 12N, определенные в метрах. Зная пространственную привязку, можно разместить эти две геометрии на поверхность земли. В результате обнаружится, что эти две геометрии фактически накладываются, даже при том, что числа, которые в них используются, абсолютно разные:



Во всех базах данных, кроме самых простых, рекомендуется сохранять пространственную привязку для каждого геообъекта непосредственно в самой базе данных. Это облегчает процесс отслеживания того, какая пространственная привязка используется в каждом геообъекте. Кроме того, это позволяет писать запросы и команды базы данных, которые учитывают пространственную привязку, а также переводить геометрии из одной пространственной привязки в другую, как того требуют ваши пространственные запросы.

К пространственным привязкам обычно обращаются, используя простое целочисленное значение, так называемый **идентификатор пространственной привязки**, англ. термин Spatial Reference Identifier (**SRID**). Хотя для представления разных пространственных привязок можно выбрать произвольные значения SRID, все же строго рекомендуется в качестве стандартных значений SRID использовать коды **Европейской нефтяной топографической группы**, англ. термин European Petroleum Survey Group (**EPSG**)[1]. Использование этого всемирно признанного стандарта делает ваши данные взаимозаменяемыми с другими базами данных и позволяет таким инструментам, как библиотеки `OGR` и `Mapnik`, идентифицировать пространственную привязку, которая используется в ваших данных.

Чтобы узнать больше о EPSG-кодах и значениях SRID в целом, обратитесь к странице http://epsg-registry.org.

PostGIS автоматически создает таблицу с именем `spatial_ref_sys`, в которой содержатся имеющиеся значения SRID. В эту таблицу предварительно загружен список из более 3000 обычно используемых пространственных привязок, все с определенным EPSG-кодом. Поскольку в этой таблице значение SRID является первичным ключом, инструменты, которые получают доступ к этой базе данных, могут обращаться к этой таблице для выполнения динамических трансформаций координат при помощи библиотеки PROJ.4.

При создании таблицы можно указать тип хранимых данных – `GEOMETRY` или `GEOGRAPHY` и используемое в этих данных значение SRID. Вот пример:

```
CREATE TABLE test (outline GEOMETRY(LINESTRING, 2193))
```

Определенная таким образом таблица будет принимать геометрии только с заданным типом и с заданной пространственной привязкой.

При вставке записи в таблицу можно тоже указывать SRID, как показано здесь:

```
INSERT INTO test (outline) VALUES (ST_GeometryFromText(wkt, 2193))
```

Хотя значение SRID необязательное, необходимо применять его повсюду, где это возможно, чтобы сообщать базе данных, какая в геометрии используется пространственная привязка. По сути, PostGIS *требует* применения корректно-

---

[1] Эта организация в настоящее время называется *Международной ассоциацией производителей нефти и газа*, англ. The International Association of Oil & Gas Producers (IOGP). IOGP ведет базу данных систем координат, которая сегодня является de facto стандартом в сфере ГИС. – *Прим. перев.*

го значения SRID, в случае если столбец настроен на использование конкретного SRID. Это защитит от случайного смешения пространственных привязок внутри таблицы.

Рекомендуем: используйте для данных приемлемую пространственную привязку

Пространственные данные, которые вы импортируете в свою базу данных, всегда находятся в конкретной пространственной привязке. Это вовсе не означает, тем не менее, что они должны *оставаться* в этой пространственной привязке. Во многих случаях эффективнее и точнее будет преобразовать ваши данные в пространственную привязку, которая наиболее подходит для ваших конкретных потребностей. Разумеется, то, что вы считаете подходящим, зависит от того, чего вы хотите достигнуть.

Когда вы используете поля с типом GEOMETRY, PostGIS предполагает, что ваши координаты спроецированы на координатную плоскость. Если вы будете использовать поле с этим типом для хранения в базе данных географических координат широты и долготы, то вы будете ограничены в том, что можно сделать. Разумеется, для сравнения двух геообъектов (например, чтобы увидеть, не пересекается ли один геообъект с другим) в базе данных можно использовать географические (неспроецированные) координаты, и вы сможете сохранять и оперативно получать геоданные. Однако все расчеты, связанные с площадями или расстояниями, будут почти бессмысленными.

Рассмотрим, например, что случится, если поручить PostGIS вычислить длину геометрии ломаной, хранимой в поле GEOMETRY:

```
# SELECT ST_Length(geom) FROM roads WHERE id=9513;
ST_Length(geom)
-----------------
0.364491142657260
```

Полученное значение «длины» находится в десятичных градусах, в которых не очень-то много пользы. Если вам нужно выполнить на своих геоданных расчет длины и площади (и, скорее всего, вам на определенном этапе это действительно понадобится сделать), то у вас есть три варианта:

- использовать для хранения данных поле GEOGRAPHY;
- перед тем как выполнить расчет расстояния или длины, преобразовать геообъекты в прямоугольные координаты;
- с самого начала хранить все свои геометрии в прямоугольных координатах.

Рассмотрим все эти варианты подробнее.

Вариант 1: использование поля GEOGRAPHY

Хотя тип поля GEOGRAPHY чрезвычайно полезен и позволяет работать непосредственно с географическими координатами, у него на самом деле есть несколько серьезных недостатков. В частности:

- расчеты на географических координатах занимают приблизительно на порядок больше времени, чем те же самые расчеты с использованием прямоугольных координат;
- тип GEOGRAPHY поддерживает значения широты и долготы только по датуму WGS84 (SRID 4326);
- ряд функций, доступных для прямоугольных координат, пока не поддерживается типом GEOGRAPHY.

Несмотря на это, вариант с полем GEOGRAPHY, возможно, стоит рассмотреть на предмет его применения.

Вариант 2: трансформация объектов по мере необходимости

Другая возможность состоит в том, чтобы хранить свои данные в географических координатах широты и долготы и переводить их в картографическую систему координат перед выполнением расчета расстояния или площади. Хотя это и сработает и даже даст вам точные результаты, тем не менее нужно делать это осмотрительно, потому что перед началом расчетов можно забыть выполнить перевод данных в картографическую систему координат. Кроме того, выполнение динамических трансформаций больших количеств геометрий очень ресурсоемкое по времени. Несмотря на эти проблемы, есть ситуации, когда хранение географических координат имеет смысл. Мы вскоре их рассмотрим.

Вариант 3: преобразование объектов с самого начала

Поскольку перевод геообъектов из одной пространственной привязки в другую довольно ресурсоемкое по времени, часто имеет смысл сделать это единожды во время импортирования своих данных и хранить их в базе данных уже переведенными в картографическую систему координат.

В таком виде вы сможете выполнять нужные пространственные расчеты быстро и точно. Однако есть ситуации, где это не самый лучший вариант, как мы убедимся в следующем разделе.

Когда использовать географические координаты

Как мы убедились в *главе 2* «*Геоинформационные системы*», перенос объектов из трехмерной поверхности Земли на двумерную координатную плоскость никогда не может быть выполнен в совершенстве. Как раз в этом и заключается математический трюизм, гласящий, что в любой проекции всегда имеются ошибки.

Для хранения таких значений, как расстояние или площадь конкретного участка поверхности Земли, обычно выбирают разные картографические проекции. Например, проекция Меркатора точна в тропиках, но искажает геообъекты ближе к полюсам.

Вследствие такого неизбежного искажения прямоугольные координаты работают лучше всего, когда ваши геоданные покрывают только часть поверхности Земли. Если вы имеете дело лишь с данными для Австрии, то картографическая система координат будет работать действительно очень хорошо. Но если ваши

данные содержат геообъекты, находящиеся и в Австрии, и в Австралии, то использование тех же самых прямоугольных координат для обоих наборов геообъектов снова приведет к неточным результатам.

По этой причине обычно лучше использовать картографическую систему координат для данных, которые покрывают только часть поверхности Земли, а географические координаты будут лучше всего работать, если приходится хранить данные, покрывающие значительные ее участки.

Конечно, использование географических координат приводит к собственным проблемам, как мы уже обсудили ранее. Вот почему рекомендуется применять именно ту пространственную привязку, которая *подходит* для ваших конкретных потребностей, а то, что для вас подходит, зависит от того, какие данные вам нужно хранить и как вы намереваетесь их использовать.

Лучший способ узнать, что подойдет лучше, – проэкспериментировать: попробуйте импортировать свои данные с использованием обеих пространственных привязок и напишите несколько тестовых программ для работы с импортированными данными. Это поможет узнать, какие пространственные привязки быстрее и легче в работе, вместо того чтобы с необходимостью прибегать к гаданию на кофейной гуще.

Рекомендуем: избегайте динамических трансформаций внутри запроса

Предположим, у вас есть таблица городов `cities` со столбцом `geom`, содержащим геометрии многоугольника `POLYGON` в координатах зоны 12N UTM (EPSG-код 32612). Будучи компетентным разработчиком геоприложений, вы настроили пространственный индекс по этому столбцу.

Далее, предположим, у вас есть переменная `pt`, которая содержит геометрию точки `POINT` в географических координатах WGS84 (EPSG-код 4326). Вы, возможно, захотите найти город, который содержит эту точку, и поэтому вы выдаете следующий ниже разумный запрос:

```
SELECT * FROM cities WHERE
  ST_Contains(ST_Transform(geom, 4326), pt);
```

В результате вы получите правильный ответ, но при этом его обработка займет чрезвычайно много времени. Почему? Потому что, прежде чем база данных сможет проверить, не находится ли точка внутри геометрии, выражение `ST_Transform(geom, 4326)` будет конвертировать геометрии *всех* многоугольников таблицы из координат UTM в зоне 12N в координаты долготы и широты WGS84. Пространственный индекс совершенно проигнорирован, так как он находится в неправильной системе координат.

Сравните со следующим ниже запросом:

```
SELECT * FROM cities WHERE
  Contains(geom, Transform(pt, 32612));
```

Совсем незначительное изменение, но результат отличается существенно. Вместо того чтобы занимать часы, ответ будет возвращен почти сразу же. Почему? Поскольку переменная pt от одной записи к другой не меняется, то выражение ST_Transform(pt, 32612) вызывается всего однажды, и потому при вызове функции ST_Contains() используется ваш пространственный индекс, который быстро находит соответствующий город.

Вывод здесь простой: надо разбираться в том, что вы поручаете сделать своей базе данных, и помнить о структурировании своих запросов, чтобы избежать динамических трансформаций больших количеств геометрий.

Рекомендуем: не создавайте геометрии внутри запроса

В рамках обсуждения запросов, которые могут заставить базу данных выполнять огромный объем работы, рассмотрим следующий ниже пример (poly обозначает многоугольник):

```
SELECT * FROM cities WHERE
  NOT ST_IsEmpty(ST_Intersection(outline, poly));
```

В некотором смысле получается совершенно разумно: определить все города, у которых результат пересечения контура города с заданным многоугольником непустой. И база данных действительно сможет ответить на этот запрос – просто это займет чрезвычайно много времени. И надеемся, вы понимаете, почему: функция ST_Intersection() создает новую геометрию из двух существующих геометрий. Это означает, что для каждой строки в таблице базы данных создается новая геометрия и затем передается в функцию ST_IsEmpty(). Как можно предположить, эти типы операций чрезвычайно медленные. Чтобы избежать создания новой геометрии на каждой итерации, можно перефразировать запрос, как показано здесь:

```
SELECT * FROM cities WHERE ST_Intersects(outline, poly);
```

Хотя этот пример может показаться очевидным, во многих случаях разработчики геопространственных приложений об этом правиле забывают и при этом удивляются, почему выполнение их запросов занимает много времени. Далее, чтобы увидеть, не находится ли точка на заданном расстоянии от многоугольника, как правило, используют функцию ST_Buffer(), как, например, здесь:

```
SELECT * FROM cities WHERE
  ST_Contains(ST_Buffer(outline, 100), pt);
```

Опять же, этот запрос будет работать, но будет делать это крайне медленно. Гораздо лучше использовать функцию ST_DWithin():

```
SELECT * FROM cities WHERE ST_DWithin(outline, pt, 100);
```

Согласно общему правилу, следует запомнить, что вы *никогда* не должны вызывать какую-либо функцию, которая возвращает объект GEOMETRY либо GEOGRAPHY, внутри подвыражения WHERE оператора SELECT.

Рекомендуем: грамотно используйте пространственные индексы

Подобно индексам в обычных базах данных, которые вносят колоссальный вклад в быстродействие и эффективность базы данных, пространственные индексы – это тоже чрезвычайно мощный инструмент ускорения ваших запросов к базе данных. Но, как и все мощные инструменты, тем не менее у них есть свои пределы:

- если пространственный индекс не определен явным образом, то база данных не сможет его использовать. И наоборот, если у вас слишком много пространственных индексов, то работа базы данных замедлится, потому что каждый индекс требует обновления всякий раз, когда запись добавляется, обновляется или удаляется. Таким образом, чрезвычайно важно определить правильный набор пространственных индексов: индексируйте только ту информацию, которую вы будете искать, и ничего больше;
- так как пространственные индексы работают на ограничительных рамках геометрий, они могут сообщать вам, только какие ограничительные рамки фактически перекрываются или пересекаются; они не смогут сообщать вам, имеют ли эту связь лежащие в их основе точки, линии или многоугольники. Таким образом, они представляют собой лишь первый этап в поиске нужной для вас информации. После того как были найдены возможные совпадения, базе данных все еще требуется поочередно проверить их геометрии;
- пространственные индексы наиболее эффективны при работе с большим количеством относительно малых геометрий. Если ваши многоугольники большие, состоящие из многих тысяч вершин, то ограничительная рамка многоугольника будет столь большой, что она пересечется с большим количеством других геометрий, и база данных должна будет вернуться к выполнению полных вычислений на многоугольнике, вместо того чтобы просто использовать ограничительную рамку. Если ваши геометрии огромны, то эти вычисления реально могут быть очень медленными – весь многоугольник должен быть загружен в память и обработан по одной вершине за одну итерацию цикла. Если такая возможность не исключается, то обычно лучше разделить крупные геометрии (в частности, большие многоугольники и составные многоугольники) на более мелкие, с тем чтобы пространственный индекс мог работать с ними эффективнее.

Рекомендуем: учитывайте пределы оптимизатора запросов своей СУБД

Когда вы отправляете запрос в СУБД, она автоматически пытается его **оптимизировать**, чтобы избежать ненужных вычислений и использовать любые доступные индексы. Например, если выдать следующий (непространственный) запрос, то СУБД будет знать, что выражение `concat("John ", "Doe")` производит константу, и поэтому вычислит его всего один раз перед выдачей запроса. Она также выполнит поиск индекса базы данных по столбцу `name` и воспользуется им, чтобы ускорить операцию.

```
SELECT * FROM people WHERE name=Concat("John ","Doe");
```

Этот тип оптимизации запросов очень мощный, и лежащая в его основе логика чрезвычайно сложная. Аналогично этому пространственные СУБД располагают своим **оптимизатором пространственных запросов**, который ради ускорения запроса ищет способы вычислить значения заранее и воспользоваться пространственными индексами. Например, рассмотрим пространственный запрос из предыдущего раздела:

```
select * from cities where ST_DWithin(outline, pt, 12.5);
```

В этом случае в функцию PostGIS `ST_DWithin()` одна геометрия передается из таблицы (`outline`), а другая представлена фиксированным значением (`pt`) наряду с нужным расстоянием (12.5 «единицы», независимо от того, что они означают в пространственной привязке данной геометрии). Оптимизатор запросов знает, как обработать этот запрос эффективным образом, заранее вычислив ограничительную рамку для фиксированной геометрии плюс нужное расстояние (`pt ±12.5`) и затем применив пространственный индекс, чтобы быстро идентифицировать записи, в которых их геометрия `outline` может лежать внутри этой расширенной ограничительной рамки.

Временами оптимизатор запросов базы данных, кажется, способен на волшебство; между тем нередки случаи, когда он невероятно глуп. Составной частью мастерства хорошего разработчика баз данных является присутствие острого чутья в отношении того, как работает оптимизатор запросов вашей СУБД, когда он не работает и что в этом случае делать.

Оптимизатор запросов PostGIS анализирует непосредственно сам запрос и смотрит на содержимое базы данных, чтобы понять, каким образом запрос можно оптимизировать. Для эффективной работы оптимизатор запросов PostGIS должен иметь актуальную статистику по содержимому базы данных. И тогда он использует изощренный генетический алгоритм, чтобы определить самый эффективный метод выполнения конкретного запроса.

В связи с этим подходом вам необходимо регулярно выполнять команду `VACUUM ANALYZE`, которая собирает статистику по базе данных, с тем чтобы оптимизатор запросов мог работать максимально эффективно. Если вы не выполняете команду `VACUUM ANALYZE`, то оптимизатор просто не сможет работать.

Вот как можно выполнить эту команду из Python:

```
import psycopg2

connection = psycopg2.connect("dbname=... user=...")
cursor = connection.cursor()

old_level = connection.isolation_level
connection.set_isolation_level(0)
cursor.execute("VACUUM ANALYZE")
connection.set_isolation_level(old_level)
```

Не переживайте по поводу логики уровня изоляции `isolation_level`; этот уровень просто позволяет выполнять команду `VACUUM ANALYZE` из Python, используя для этого трансакционно-ориентированный адаптер `psycopg2`.

Существует возможность настроить работу служебного процесса «autovacuum daemon», который срабатывает автоматически через установленные промежутки времени или после того, как содержимое таблицы изменилось достаточно, чтобы появилось основание для срабатывания. Настройка этого служебного процесса выходит за рамки данной книги.

После выполнения команды VACUUM ANALYZE оптимизатор запросов сможет приступить к оптимизации ваших запросов. Чтобы увидеть, как оптимизатор запросов работает, можно воспользоваться командой EXPLAIN SELECT. Например:

```
psql> EXPLAIN SELECT * FROM cities
         WHERE ST_Contains(geom,pt);

                      QUERY PLAN
-----------------------------------------------------
Seq Scan on cities (cost=0.00..7.51 rows=1 width=2619)
  Filter: ((geom &&
'0101000000000000000000000000000000000000000'::geometry) AND _st_
contains(geom, '0101000000000000000000000000000000000000000'::geometry))
(2 rows)
```

Не переживайте по поводу фрагмента Seq Scan; в этой таблице всего несколько записей, поэтому PostGIS знает, что может просканировать всю таблицу быстрее, чем это можно сделать посредством индекса. Когда база данных станет больше, она автоматически начнет использовать индекс, чтобы быстро идентифицировать нужные записи.

Фрагмент cost= обозначает, во что этот запрос обойдется, при этом затраты измеряются в произвольных единицах, которые по умолчанию вычисляются относительно того, сколько времени потребуется, чтобы прочитать страницу данных с диска. Два числа обозначают затраты на запуск (сколько времени потребуется, чтобы обработать первую строку) и оценочные общие затраты (сколько времени потребуется, чтобы обработать все записи в таблице). Поскольку чтение страницы данных с диска выполняется довольно быстро, общие затраты 7.51 на самом деле очень небольшие.

Самая интересная часть данного объяснения – это Filter. Приглядимся поближе к тому, что команда EXPLAIN SELECT сообщает нам о том, как PostGIS отфильтрует этот запрос. Рассмотрим первую часть:

```
(geom && '0101000000000000000000000000000000000000000'::geometry)
```

Здесь используется оператор &&, который ищет совпадающие записи, используя ограничительную рамку, заданную в пространственном индексе. Теперь рассмотрим вторую часть условия фильтра:

```
_st_contains(geom,
'0101000000000000000000000000000000000000000'::geometry)
```

Здесь используется функция ST_Contains(), которая идентифицирует правильные геометрии, содержащие нужную точку. Этот двухступенчатый процесс (сна-

чала фильтрация посредством ограничительной рамки, затем посредством самой геометрии) позволяет базе данных использовать пространственный индекс, чтобы идентифицировать записи на основе их ограничительных рамок и затем проверить потенциальные совпадения, выполняя исчерпывающий просмотр непосредственно самой геометрии. Это чрезвычайно эффективно, и, как можете убедиться, PostGIS делает это для нас автоматически, приводя к быстрому и одновременно точному поиску геометрий, которые содержат заданную точку.

Заключение

В этой главе мы провели глубокий анализ концепции хранения пространственных данных в базе данных при помощи общедоступного инструмента – расширения PostGIS. Мы узнали, что пространственные СУБД отличаются от обычных реляционных тем, что они напрямую поддерживают пространственные типы данных и используют пространственные индексы, чтобы выполнять запросы и соединения на пространственных данных. Мы увидели, что пространственные индексы используют ограничительные рамки геометрий, чтобы быстро сравнивать и читать геометрии на основе их положения в пространстве.

Затем мы рассмотрели геопространственное расширение PostGIS для СУБД PostgreSQL и приемы использования адаптера СУБД PostgreSQL `psycopg2` для получения доступа к пространственным базам данных PostGIS на основе языка Python. После установки необходимого программного обеспечения мы сконфигурировали пространственную базу данных и использовали адаптер `psycopg2` для создания необходимых таблиц базы данных, импортирования ряда пространственных данных и выполнения полезных запросов к этим данным.

Затем мы рассмотрели некоторые наиболее успешные практические приемы работы с пространственными базами данных. Мы увидели, насколько важно наряду с самими данными хранить идентификатор пространственной привязки, и рассмотрели приемы выбора пространственной привязки, которая подходит для приложения наилучшим образом.

Затем мы рассмотрели некоторые ошибки, которые могут снизить производительность базы геоданных, в том числе создание геометрий и выполнение трансформаций на лету (динамически) и использование пространственных индексов ненадлежащим образом, что в итоге не дает базе данных их использовать.

Наконец, мы узнали об оптимизаторе запросов PostGIS и о том, как использовать команду `EXPLAIN`, чтобы ясно понимать, как PostGIS будет выполнять пространственный запрос.

В следующей главе мы научимся пользоваться библиотекой Mapnik для преобразования исходных геоданных в красивые изображения географических карт.

Глава 7

Генерирование карт при помощи Python и библиотеки Mapnik

Поскольку в пространственных данных практически невозможно разобраться, пока они не будут выведены на экран компьютера, использование карт для визуального представления пространственных данных является чрезвычайно важной темой. В этой главе мы рассмотрим мощную библиотеку Python **Mapnik** для преобразования геоданных в привлекательные карты. В частности, мы рассмотрим:

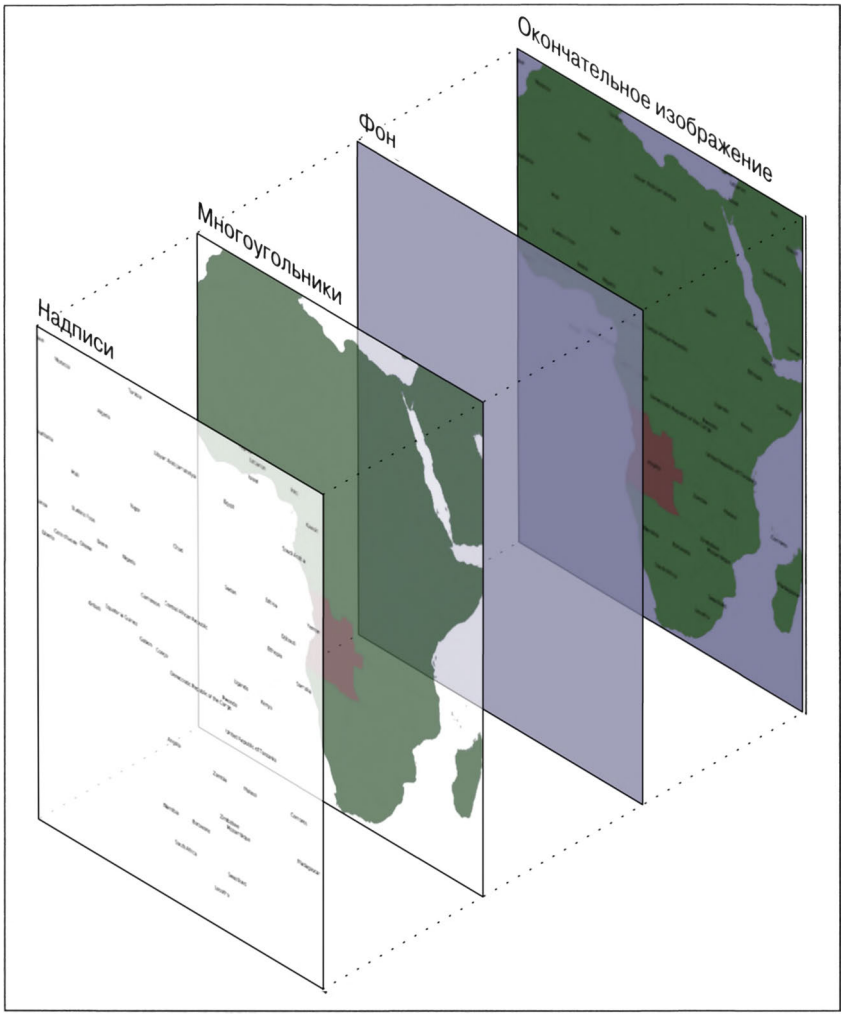
- базовые понятия библиотеки Mapnik, используемые при картографировании;
- пример программы, использующей библиотеку Mapnik для построения простой карты;
- различные источники данных, которые можно использовать для добавления геоданных в карту;
- как использовать правила, фильтры и стили для управления процессом картографирования;
- как использовать «символизаторы» для добавления к картам точек, линий, многоугольников, надписей и растровых изображений;
- каким образом карты и слои карты взаимодействуют друг с другом при построении карты;
- способы преобразования цифровой карты в графический файл.

Введение в библиотеку Mapnik

Ранее мы уже обращались к библиотеке Mapnik в *главе 3 «Библиотеки Python для геопрограммирования»*. Если вы еще этого не сделали, пожалуйста, вернитесь к разделу Mapnik указанной главы и следуйте инструкциям по ее установке на свой компьютер.

Mapnik – сложная библиотека, состоящая из нескольких разных компонентов, где среди различных названий и понятий легко запутаться. Начнем наше исследование библиотеки Mapnik, взглянув на простую карту:

Не сразу бросается в глаза только то, что различные элементы внутри карты нанесены слоями, как показано ниже:



Для того чтобы сгенерировать эту карту, необходимо поручить библиотеке Mapnik сначала нарисовать фон, затем нанести многоугольники и наконец надписи. В результате многоугольники гарантированно разместятся поверх фона, а надписи появятся перед многоугольниками и перед фоном.

Строго говоря, фон не является слоем. Это просто цвет либо графическое изображение, которые библиотека наносит на карту, перед тем как она начнет отрисовку первого слоя.

Библиотека Mapnik позволяет управлять последовательностью отрисовки элементов карты посредством использования объектов **Layer** (Слой). Простая карта может состоять всего из одного слоя, однако у большинства карт имеется несколь-

ко слоев. Слои наносятся строго в обратном порядке, поэтому слой, определенный первым, появится позади всех остальных. В только что приведенном примере карты слой прямоугольников **Polygon** задан первым, за которым следует слой надписей **Label**. В результате надписи гарантированно появятся перед многоугольниками. Этот метод разделения на слои называется **алгоритмом живописца** в силу его схожести с тем, как художник наносит слои краски на холст.

Каждый слой Layer имеет свой собственный **источник данных**, который сообщает Mapnik, откуда загружать данные. Источник данных может ссылаться на файл фигур, базу пространственных данных, файл с растровым изображением либо на ряд других источников геоданных. В большинстве случаев настройка источника данных для слоя Layer очень простая.

Внутри каждого слоя визуализация геоданных контролируется посредством так называемого **символизатора**. В библиотеке Mapnik имеется много символизаторов разных типов, вместе с тем три из них представляют здесь для нас интерес:

- символизатор многоугольников `PolygonSymbolizer` используется для отрисовки заполненных многоугольников:

- символизатор линий `LineSymbolizer` используется для отрисовки контура многоугольников, а также ломаных и других линейных объектов:

- символизатор текста `Textsymbolizer` используется для отрисовки надписей и другого текста на карте.

Во многих случаях этих трех символизаторов будет достаточно, чтобы нарисовать всю карту. Действительно, почти вся ранее приведенная карта была сгенерирована при помощи всего одного символизатора многоугольников `PolygonSymbolizer`, одного символизатора линий `LineSymbolizer` и одного символизатора текста `Textsymbolizer`, как показано ниже:

Слой «Polygon»

Символизатор многоугольников

```
fill = mapnik.Color("#406040")
```

Символизатор линий

```
stroke = mapnik.Stroke(mapnik.Color("#000000"), 0.1)
```

Слой «Labels»

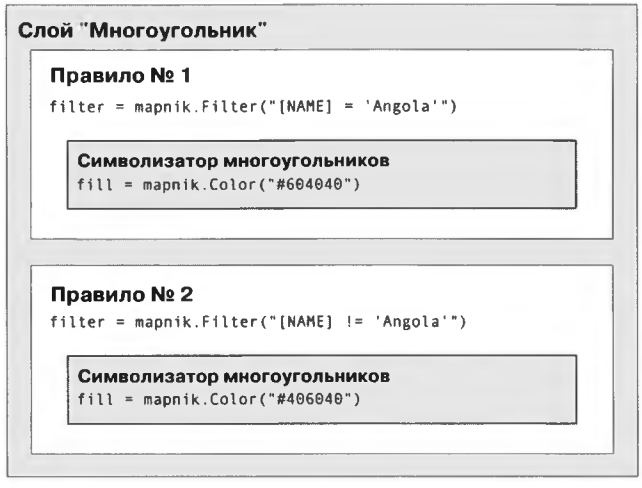
Символизатор текста

```
expression = "[NAME]"
font       = "DejaVu Sans Book"
size       = 10
color      = mapnik.Color("#000000")
```

Внутри каждого слоя символизаторы обрабатываются при помощи того же самого алгоритма живописца, который был описан ранее. В случае со слоем многоугольников Polygon поверх символизатора многоугольников `PolygonSymbolizer` задействован символизатор линий `LineSymbolizer`.

Отметим, что символизаторы связаны со слоем не напрямую, а скорее *косвенно* ассоциативной связью при помощи стилей и правил. Мы обратимся к стилям через минуту, а сейчас приглядимся поближе к используемому в библиотеке Mapnik понятию правил.

Правило позволяет применять набор символизаторов, только когда удовлетворяется заданное условие. Например, на карте, приведенной в начале этой главы, Ангола была изображена другим цветом. Это было сделано путем определения в слое многоугольников Polygon двух правил, как показано ниже:



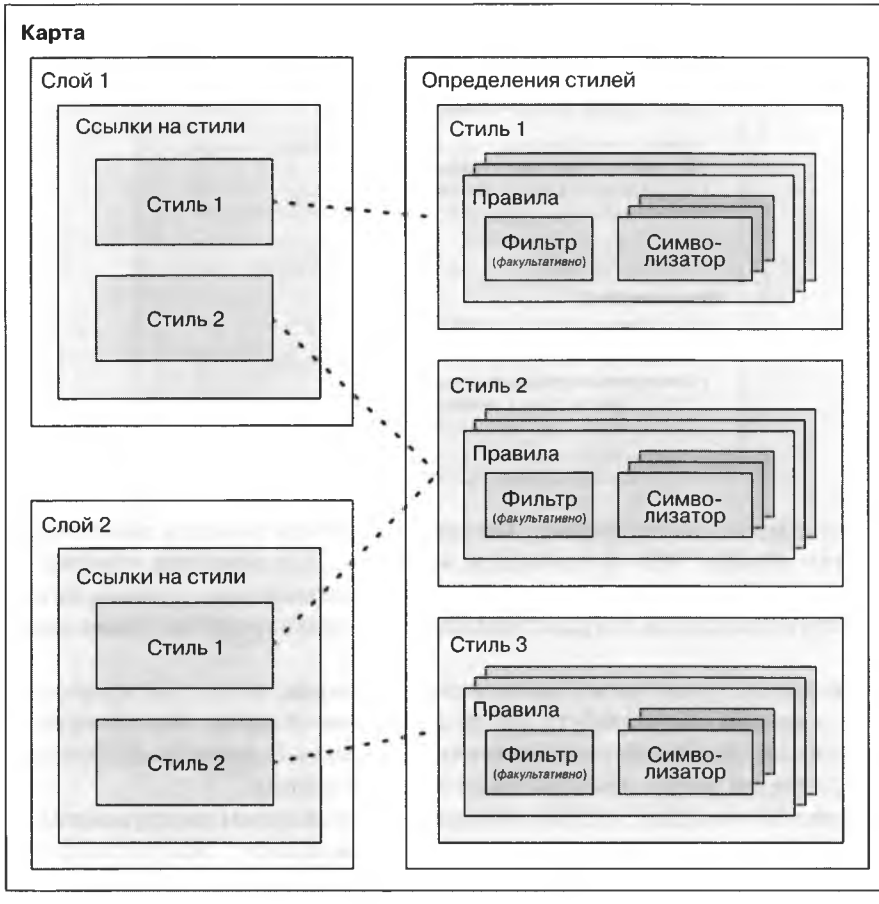
Первое правило имеет **фильтр**, который относится только к объектам, у которых есть атрибут `NAME` со значением «Angola». Для объектов, которые соответствуют этому условию фильтрации, будет использовано правило символизатора многоугольников PolygonSymbolizer, которое окрасит их темно-красным цветом.

Второе правило располагает аналогичным фильтром, на этот раз проверяющим объекты, у которых нет атрибута `NAME` со значением «Angola». Эти объекты отрисованы при помощи символизатора многоугольников PolygonSymbolizer второго правила, который рисует объекты темно-зеленым цветом.

Совершенно очевидно, правила являются очень мощным инструментом выборочной корректировки метода нанесения объектов на карту. Мы рассмотрим правила более подробно в разделе «*Правила, фильтры и стили*» этой главы.

Когда вы задаете свой символизатор, вы размещаете его в правилах. Сами правила сгруппированы в **стили**, при помощи которых ваши правила можно организовывать и отслеживать. Каждый слой карты имеет список стилей, которые применяются к этому конкретному слою.

Хотя связь между слоями, стилями, правилами, фильтрами и символизаторами может показаться замысловатой, тем не менее она обеспечивает значительную часть мощи и гибкости библиотеки Mapnik. Важно, чтобы вы понимали, как эти различные классы взаимодействуют между собой:



Как видите, определения стилей хранятся непосредственно внутри карты, при этом в различных слоях используются ссылки на стили, чтобы идентифицировать, какой из определенных стилей будет использоваться в каждом слое. Это работает почти так же, как таблица стилей в документе системы обработки текстов, где вы определяете стили и потом пользуетесь ими вновь и вновь. Отметим, что тот же самый стиль можно использовать многократно.

Библиотека Mapnik обеспечивает альтернативный способ определения своих собственных всевозможных стилей посредством использования файла определений цифровой карты (map definition) в XML-формате. Правда, в этой книге мы не будем пользоваться файлами определений цифровой карты, так как файлы в формате XML довольно трудно читать. К тому же легче создать свои определения непосредственно при помощи программного кода на Python[1].

[1] В главах 8, 9 и 13 приведены примеры реализации на основе XML. – *Прим. перев.*

Создание образца карты

Чтобы лучше понять, каким образом различные части библиотеки Mapnik взаимодействуют между собой, напишем простую программу на Python, чтобы сгенерировать карту, показанную в начале этой главы. В этой карте используется набор данных границ стран мира, который вы скачали в одной из предыдущих глав; скопируйте каталог TM_WORLD_BORDERS-0.3 с файлами фигур в удобное место и создайте там же новый сценарий Python. Назовем эту программу createExampleMap.py[1].

Начнем с импорта инструментария библиотеки Mapnik и определения нескольких констант, которые потребуются программе:

```
import mapnik
MIN_LAT = -35
MAX_LAT = +35
MIN_LONG = -12
MAX_LONG = +50
MAP_WIDTH = 700
MAP_HEIGHT = 800
```

Константы MIN_LAT, MAX_LAT, MIN_LONG и MAX_LONG определяют географические координаты широты и долготы для части нашей планеты, которая будет отображаться на карте, тогда как константы MAP_WIDTH и MAP_HEIGHT задают размер генерируемого изображения цифровой карты в пикселах. Разумеется, по желанию их можно изменить.

Теперь мы готовы определить содержимое карты. Эта карта будет иметь два слоя, один для отрисовки многоугольников и другой для нанесения надписей. Мы определим объект Style для каждого из этих двух слоев. Начнем со стиля для слоя с многоугольными объектами Polygons:

```
polygonStyle = mapnik.Style()
```

Как уже обсуждалось в предыдущем разделе, объект Filter позволяет выбирать, к каким конкретным объектам правило будет применяться. В этом случае нам нужно установить два правила: одно, чтобы нарисовать Анголу темно-красным цветом, а другое, чтобы нарисовать все остальные страны темно-зеленым:

```
rule = mapnik.Rule()
rule.filter = mapnik.Filter("[NAME] = 'Angola'")
symbol = mapnik.PolygonSymbolizer(mapnik.Color("#604040"))
rule.symbols.append(symbol)

polygonStyle.rules.append(rule)

rule = mapnik.Rule()
rule.filter = mapnik.Filter("[NAME] != 'Angola'")
symbol = mapnik.PolygonSymbolizer()
```

[1] Этот сценарий выполняется только в среде Python 2 (в данном случае 2.7.12). См. настройки среды Spyder в *«Комментарии переводчика»*. – *Прим. перев.*

```
symbol.fill = mapnik.Color("#604040")
rule.symbols.append(symbol)

polygonStyle.rules.append(rule)
```

Отметим то, как мы используем символизатор многоугольников PolygonSymbolizer, чтобы заполнить контур каждой страны соответствующим цветом, и затем добавляем этот символизатор в наше текущее правило. Мы добавляем правила в наш стиль для многоугольников по мере определения правил.

Заполнив многоугольники стран цветом, теперь определим дополнительное правило отрисовки контуров многоугольников:

```
rule = mapnik.Rule()
symbol = mapnik.LineSymbolizer(mapnik.Color("#000000"), 0.1)
rule.symbols.append(symbol)

polygonStyle.rules.append(rule)
```

Вот и все, что требуется, чтобы отобразить многоугольники стран на карте. Теперь пойдем дальше и определим второй объект Style для слоя надписей Labels:

```
labelStyle = mapnik.Style()

rule = mapnik.Rule()
symbol = mapnik.TextSymbolizer(mapnik.Expression("[NAME]"),
                               "DejaVu Sans Book", 12,
                               mapnik.Color("#000000"))
rule.symbols.append(symbol)

labelStyle.rules.append(rule)
```

Для нанесения надписей на карту в этом стиле используется символизатор текста TextSymbolizer. Отметим, что мы создаем объект Expression (Выражение), чтобы определить отображаемый текст, – в этом случае мы хотим отобразить атрибут NAME файла фигур; как и следовало ожидать, этот атрибут содержит название страны.

В этом примере мы используем всего по одному стилю в каждом слое. Генерируя более сложную карту, вам, как правило, понадобится много стилей, которые могут применяться к каждому слою, при этом по мере надобности стили могут использоваться в разных слоях совместно. В этом примере, тем не менее, мы стараемся сохранить определение карты максимально простым.

После настройки стилей можно начинать определять слои карты. Правда, перед этим мы должны назначить источник данных:

```
datasource = mapnik.Shapefile(file="TM_WORLD_BORDERS-0.3/" +
                               "TM_WORLD_BORDERS-0.3.shp")
```

Затем мы можем определить два слоя, которые используются на нашей карте:

```
polygonLayer = mapnik.Layer("Polygons")
polygonLayer.datasource = datasource
polygonLayer.styles.append("PolygonStyle")

labelLayer = mapnik.Layer("Labels")
labelLayer.datasource = datasource
labelLayer.styles.append("LabelStyle")
```

Отметим, что, вместо того чтобы вставлять стиль непосредственно, мы обращаемся к стилям по имени. Это позволяет использовать конкретный стиль для нескольких слоев. Чуть позже мы добавим в нашу карту определения стилей.

Наконец, теперь можно создать наш объект `Map`. Объект `Map` библиотеки Mapnik имеет ширину и высоту, проекцию, цвет фона, список стилей и список слоев, которые образуют карту:

```
map = mapnik.Map(MAP_WIDTH, MAP_HEIGHT,
                 "+proj=longlat +datum=WGS84")
map.background = mapnik.Color("#8080a0")

map.append_style("PolygonStyle", polygonStyle)
map.append_style("LabelStyle", labelStyle)

map.layers.append(polygonLayer)
map.layers.append(labelLayer)
```

На завершающем этапе потребуется всего лишь поручить библиотеке Mapnik сделать увеличение нужного региона мира и затем перенести карту в графический файл:

```
map.zoom_to_box(mapnik.Box2d(MIN_LONG, MIN_LAT,
                             MAX_LONG, MAX_LAT))
mapnik.render_to_file(map, "createExampleMap_map.png")
```

Если вы выполните эту программу и откроете получившийся файл `createExampleMap_map.png`, то увидите сгенерированную вами карту:

Для сравнения ниже приводится декларативная реализация данной программы, которая будет работать в Python 3, с использованием XML:

```
# createExampleMapXML1.py
import mapnik

MIN_LAT  = -35
MAX_LAT  = +35
MIN_LONG = -12
```

```
MAX_LONG = +50

MAP_WIDTH  = 700
MAP_HEIGHT = 800

map_output = '../images/ExampleMap1.png'

xml = '''<?xml version="1.0" encoding="utf-8"?>
<Map background-color="#8080a0" srs="+proj=latlong +datum=WGS84">
    <Style name="PolygonStyle">
        <Rule>
            <Filter>[NAME] = 'Angola'</Filter>
            <PolygonSymbolizer fill="#604040" />
        </Rule>
        <Rule>
            <Filter>[NAME] != 'Angola'</Filter>
            <PolygonSymbolizer fill="#406040" />
        </Rule>
        <Rule>
            <LineSymbolizer stroke="#000000" stroke-width="0.1" />
        </Rule>
    </Style>
    <Style name="LabelStyle">
        <Rule>
            <TextSymbolizer face-name="DejaVu Sans Book" size="12" fill="#000000">
                [NAME]
            </TextSymbolizer>
        </Rule>
    </Style>

     <Layer name="Polygons" srs="+proj=latlong +datum=WGS84">
         <StyleName>PolygonStyle</StyleName>
         <Datasource>
            <Parameter name="file">../data/TM_WORLD_BORDERS-0.3/TM_WORLD_BORDERS-
0.3.shp</Parameter>
            <Parameter name="type">shape</Parameter>
         </Datasource>
     </Layer>
     <Layer name="Labels" srs="+proj=latlong +datum=WGS84">
         <StyleName>LabelStyle</StyleName>
         <Datasource>
            <Parameter name="file">../data/TM_WORLD_BORDERS-0.3/TM_WORLD_BORDERS-
0.3.shp</Parameter>
            <Parameter name="type">shape</Parameter>
         </Datasource>
     </Layer>
</Map>
'''

gmap = mapnik.Map(MAP_WIDTH, MAP_HEIGHT)
mapnik.load_map_from_string(gmap, xml)
```

```
bbox = mapnik.Box2d(MIN_LONG, MIN_LAT, MAX_LONG, MAX_LAT)
# то же самое
# bbox = (mapnik.Envelope( MIN_LONG, MIN_LAT, MAX_LONG, MAX_LAT ))

gmap.zoom_to_box(bbox)
mapnik.render_to_file(gmap, map_output)
```

Разумеется, с помощью библиотеки Mapnik можно делать гораздо больше, но этот пример отвечает на основные вопросы, и его должно быть достаточно, чтобы дать вам возможность генерировать свои собственные карты. Убедитесь, что вы в достаточной мере поработали с этим примером, чтобы поближе познакомиться с тем, как функционирует эта библиотека. Вот некоторое из того, что вы бы могли попробовать:

- корректировка констант MIN_LAT, MIN_LONG, MAX_LAT и MAX_LONG в начале программы, чтобы приблизить страну, в которой вы проживаете;
- изменение размера генерируемого изображения;
- изменение цвета карты;
- добавление новых правил для отображения названия страны в разных размерах и цвете шрифта, исходя из населения страны.

Для этого вам потребуется определить фильтры, которые будут выглядеть следующим образом:

```
mapnik.Filter("[POP2005] > 1000000 and [POP2005] <= 2000000")
```

Понятия библиотеки Mapnik

В этом разделе мы более подробно займемся исследованием питоновского интерфейса к инструментам библиотеки Mapnik. Поскольку функционал библиотеки слишком обширен, чтобы охватить его в полном объеме, мы сконцентрируемся на самых важных аспектах библиотеки Mapnik, которые вы, вероятно, будете использовать в своих собственных программах. Не стесняйтесь обращаться к документации по библиотеке Mapnik (http://mapnik.org/docs) в отношении того, что не было охвачено в этой главе.

Источники данных

Объект источника данных служит «мостом» между библиотекой Mapnik и вашими геоданными. Вы обычно создаете источник данных, используя один из следующих вспомогательных конструкторов, и затем добавляете этот источник данных к любым объектам-слоям Layer, в которых эти данные используются:

```
layer.datasource = datasource
```

Библиотека Mapnik поддерживает большое количество различных типов источников данных, правда, некоторые из них являются экспериментальными либо обращаются к данным в коммерческих базах данных. Приглядимся поближе к типам источников данных, которые вы, скорее всего, найдете полезными.

Источник данных Shapefile

В качестве источника данных библиотеки Mapnik очень просто использовать файл фигур. Вспомогательному конструктору `mapnik.Shapefile()` требуется всего лишь предоставить имя и путь к каталогу нужного файла фигур:

```
datasource = mapnik.Shapefile(file="путь/к/shapefile.shp")
```

При помощи необязательного параметра `encoding` можно указывать символьную кодировку для использования в атрибутах файла фигур, как, например, здесь:

```
datasource = mapnik.Shapefile(file="shapefile.shp",
                              encoding="latin1")
```

Если параметр кодировки не задан, то предполагается, что в файле фигур используется символьная кодировка UTF-8.

Источник данных PostGIS

Источник данных PostGIS позволяет включать данные из базы данных PostGIS:

```
datasource = mapnik.PostGIS(user="..." password="...",
                            dbname="...", table="...")
```

Отметим, что вы предоставляете имя пользователя и пароль доступа к базе данных PostGIS, имя базы данных и название таблицы, содержащей пространственные данные, которые вы хотите нанести на вашу карту.

При получении данных из базы данных PostGIS следует учитывать несколько проблем, связанных с производительностью. При использовании объекта `mapnik.Filter` для выборочного отображения объектов, взятых из большой таблицы базы данных, процесс генерирования вашей карты может занять много времени. Это объясняется тем, что библиотеке Mapnik приходится по очереди загружать каждый объект таблицы базы данных, проверяя его на совпадение с этим выражением фильтрации. Намного более эффективный прием состоит в применении так называемого **запроса с подвыборкой** (с подзапросом), который ограничивает набор данных, получаемый библиотекой Mapnik из базы данных.

Чтобы использовать подзапрос Select, вы заменяете имя таблицы оператором SQL `SELECT`, который выполняет фильтрацию и возвращает информацию, необходимую Mapnik, чтобы сгенерировать слой карты. Вот пример:

```
query = "(SELECT geom FROM myBigTable WHERE level=1) AS data"
datasource = mapnik.PostGIS(user="...", password="...",
                            dbname="...", table=query)
```

Мы заменили имя таблицы подзапросом PostGIS, который фильтрует все записи со значением уровня `level`, равным 1, и для совпадающих записей возвращает только поле `geom`. с типом `GEOMETRY`.

Отметим, что если вы используете подзапрос, то важно указывать все поля, к которым вы хотите обратиться в вашем операторе `SELECT`. Если вы не укажете

какое-либо поле в элементе SELECT подзапроса, то оно не будет доступно для использования библиотекой Mapnik.

Источник данных Gdal

Источник данных Gdal позволяет вам включать в свою карту любой совместимый с библиотекой GDAL файл данных с растровым изображением. Его применение достаточно прямолинейное:

```
datasource = mapnik.Gdal(file="myRasterImage.tiff")
```

Когда имеется источник данных Gdal, для нанесения его на карту можно воспользоваться символизатором растра RasterSymbolizer:

```
layer = mapnik.Layer("myLayer")
layer.datasource = datasource
layer.styles.append("myLayerStyle")

symbol = mapnik.RasterSymbolizer()

rule = mapnik.Rule()
rule.symbols.append(symbol)

style = mapnik.Style()
style.rules.append(rule)

map.append_style("myLayerStyle", style)
```

Источник данных MemoryDatasource

Буферный источник данных MemoryDatasource позволяет вам вручную определять геоданные, которые появляются на карте. Чтобы воспользоваться буферным источником данных MemoryDatasource, сначала создается сам объект источника данных:

```
datasource = mapnik.MemoryDatasource()
```

Затем определяется контекстный объект библиотеки Mapnik mapnik.Context, в котором перечисляются те атрибуты, которые вы хотите связать с каждым геообъектом источника данных:

```
context = mapnik.Context()
context.push("NAME")
context.push("ELEVATION")
```

Затем для каждого геообъекта, который вы хотите разместить на карте, вы создаете объект mapnik.Feature:

```
feature = mapnik.Feature(context, id)
```

Отметим, что идентификационный параметр id должен иметь уникальное целочисленное значение, которое будет определять этот геообъект внутри буферного источника данных.

После того как объект Feature создан, вы затем устанавливаете его атрибуты. Это делается аналогично тому, как вы определяете записи внутри словаря Python:

```
feature['NAME'] = "Hawkins Hill"
feature['ELEVATION'] = 1624
```

Эти атрибуты могут использоваться правилами с целью выбора тех геообъектов, которые должны быть нанесены на карту, а также символизатором текста `TextSymbolizer` для нанесения значения атрибута на карту.

С каждым объектом `Feature` может быть связана одна или несколько геометрий. Самый легкий способ назначить геометрию объекту заключается в применении метода `add_geometries_from_wkt()`, как показано ниже:

```
feature.add_geometries_from_wkt("POINT (174.73 -41.33)")
```

В заключение при помощи метода `add_feature()` вы добавляете объект `Feature` в буферный источник данных MemoryDatasource:

```
datasource.add_feature(feature)
```

После настройки буферного источника данных `MemoryDatasource` можно добавить его к слою карты и воспользоваться любым векторным символизатором, чтобы нанести геообъекты на карту.

Правила, фильтры и стили

Как мы уже убедились ранее в этой главе, в библиотеке Mapnik правила используются для того, чтобы указывать, какой именно символизатор будет использоваться для генерирования заданного геообъекта. Правила группируются в стиль, а всевозможные стили добавляются в вашу карту, после чего к ним обращаются по имени, когда вы выполняете настройку слоя карты. В этом разделе мы займемся исследованием связей между правилами, фильтрами и стилями и увидим, что именно можно сделать с этими разными классами библиотеки Mapnik.

Приглядимся поближе к классу `Rule` библиотеки Mapnik. Правило состоит из двух частей: набора условий и списка символизаторов. Если условия правила удовлетворяются, то для нанесения совпадающих геообъектов на карту используются символизаторы.

Правило библиотеки Mapnik поддерживает два основных типа условий:

- при помощи **фильтра** можно определять выражение, которому геообъект должен удовлетворять, чтобы быть нанесенным на карту;
- правило может иметь итоговое условие **else** (иначе), означающее, что правило будет применено, только если ни одно условие правил в стиле не было удовлетворено.

Если условия правила были удовлетворены, то для нанесения геообъекта на карту будет применен ассоциированный с этим правилом список символизаторов.

Рассмотрим фильтры и условия else подробнее.

Фильтры

Конструктор `Filter()` принимает единственный строковый параметр, содержащий выражение, с которым геообъект должен совпадать, чтобы правило было

применено. Затем вы сохраняете возвращенный объект `Filter()` в атрибуте `filter` этого правила:

```
rule.filter = mapnik.Filter("...")
```

Возьмем очень простое выражение фильтрации, сравнивающее поле или атрибут с конкретным значением:

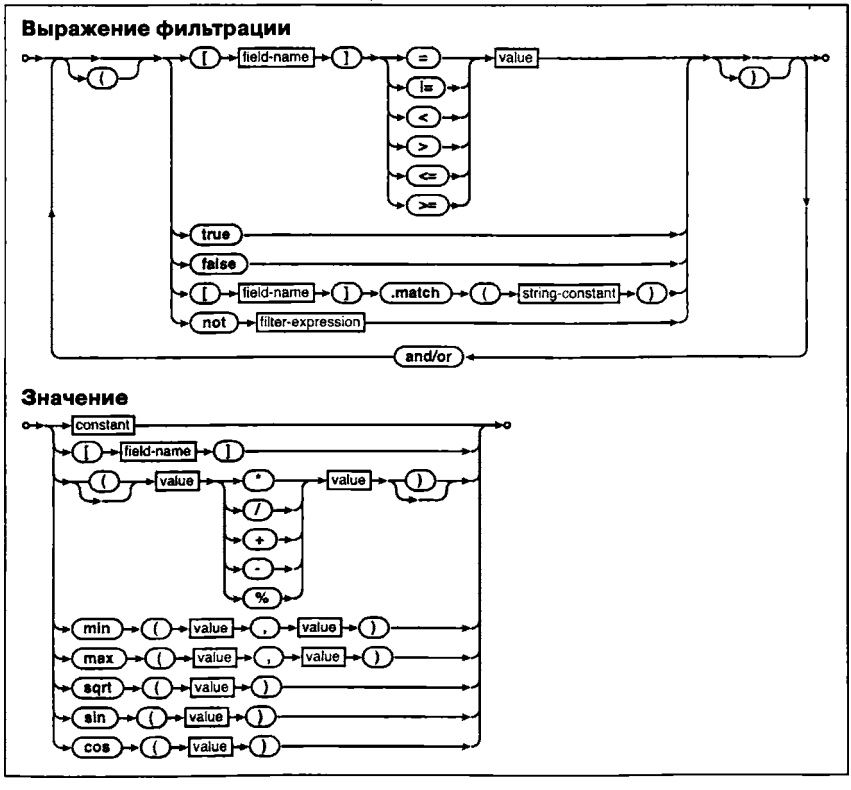
```
filter = mapnik.Filter("[level] = 1")
```

Сравнение строковых значений можно выполнять, поместив символ одинарной кавычки с обеих сторон значения, как показано тут:

```
filter = mapnik.Filter("[type] = 'CITY'")
```

Отметим, что имя поля и значение чувствительны к регистру и что необходимо ограничить поле или название атрибута квадратными скобками.

Конечно, сопоставление поля со значением является самым элементарным возможным типом сравнения. Выражения фильтрации имеют свой собственный мощный и гибкий синтаксис для определения условий, аналогичных по концепции выражению языка SQL `WHERE`. Следующая синтаксическая диаграмма описывает все варианты написания символьных последовательностей выражения фильтрации:



Кроме того, библиотека Mapnik также позволяет выполнять фильтрацию на основе типа геометрии, используя для этого следующую специальную синтаксическую конструкцию:

```
filter = mapnik.Filter("[mapnik::geometry_type] = point")
```

Этот вид выражения фильтрации поддерживает следующие значения типа геометрии:

- point (точка);
- linestring (ломаная);
- polygon (многоугольник);
- collection (коллекция).

Правила с итоговым условием «иначе»

Предположим, вы хотите отобразить некоторые объекты в одном цвете, а все остальные – в другом. Один из приемов достигнуть этого при помощи фильтров Mapnik состоит в следующем:

```
rule1.filter = mapnik.Filter("[level] = 1")
...
rule2.filter = mapnik.Filter("[level] != 1")
```

Это хорошо работает на простых выражениях фильтрации, но когда выражения становятся сложнее, намного легче использовать правило с итоговым условием «иначе», как показано ниже:

```
rule.set_else(True)
```

Если вы для правила вызываете set_else(True), то это правило будет использовано, если и только если условия фильтрации ни одного из предыдущих правил внутри того же самого стиля не были удовлетворены.

Правила с итоговым условием особенно полезны, если у вас много условий фильтрации и в конце вы хотите получить всеобъемлющее правило. Это всеобъемлющее правило будет применяться, если для отображения геообъекта никакое другое правило не было использовано, как в примере ниже:

```
rule1.filter = mapnik.Filter("[type] = 'city'")
rule2.filter = mapnik.Filter("[type] = 'town'")
rule3.filter = mapnik.Filter("[type] = 'village'")
rule4.filter.set_else(True)
```

Стили

Стиль библиотеки Mapnik – это просто список правил. Чтобы определить стиль, создайте новый объект mapnik.Style и по очереди добавляйте к нему правила, как показано ниже:

```
style = mapnik.Style()
style.rules.append(rule1)
style.rules.append(rule2)
...
```

Правила обрабатываются одно за другим для каждого объекта, при этом, чтобы убедиться, что геообъект соответствует правилу, условия каждого правила проверяются по очереди. Если геообъект совпадает, то связанные с правилом символизаторы будут использованы для его отображения. Если ни одно из правил с ним не совпадает, то геообъект не отображается.

После создания объекта `Style` вы добавляете в свой объект `mapnik.Map` стиль, назначив ему имя. И далее вы обращаетесь к этому стилю по имени из своих слоев карты. Например:

```
map.append_style("style_name", style)
layer.styles.append("style_name")
```

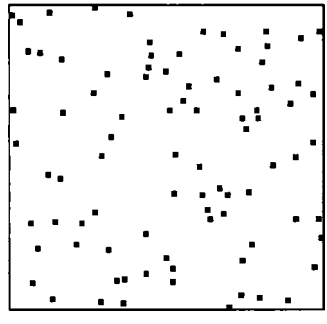
Символизаторы

Символизаторы используются для нанесения геообъектов на карту. В этом разделе мы рассмотрим ряд самых распространенных типов символизаторов, которые используются для нанесения на карту точек, линий, многоугольников, текста и растровых изображений.

Нанесение точек

Для нанесения геометрии точки вы используете **символизатор точек** `PointSymbolizer`. Он выводит изображение в указанной точке. Конструктор по умолчанию не принимает параметров и выводит каждую точку как черный квадрат размером 4×4 пиксела:

```
symbolizer = PointSymbolizer()
```



Как вариант можно предоставить путь к графическому файлу, который будет использован символизатором точек `PointSymbolizer` для нанесения каждой точки:

```
path = mapnik.PathExpression("путь/к/image.png")
symbolizer = PointSymbolizer(path)
```

Поведение символизатора точек `PointSymbolizer` можно изменить, установив следующие атрибуты:

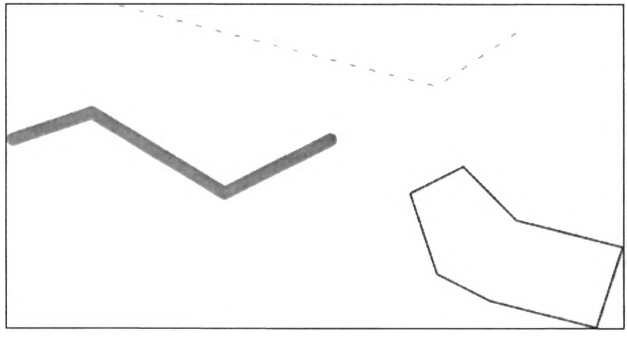
- `symbolizer.allow_overlap = True`: если вы установите этот атрибут в `True`, то будут нанесены все точки, даже если изображения налагаются друг на друга.

По умолчанию (False) подразумевается, что точки будут нанесены, только если они взаимно не накладываются;

- symbolizer.opacity = 0.75: этот атрибут управляет величиной прозрачности или транспарентности, которая используется во время отрисовки изображения. Значение 1.0 (по умолчанию) нарисует абсолютно непрозрачное изображение, тогда как значение 0.0 сделает его абсолютно прозрачным;
- symbolizer.transform = "...": представляет собой выражение SVG-преобразования, которое можно использовать для управления выводимым изображением. Например, transform="rotate(45) scale(0.5, 0.5)" повернет изображение по часовой стрелке на 45° и затем пропорционально уменьшит его до 50% от его первоначального размера.

Нанесение линий

Символизатор линий LineSymbolizer наносит линейные объекты и выполняет трассировку вокруг контура многоугольников, как показано на следующем ниже рисунке:



Символизатор линий LineSymbolizer – один из самых полезных символизаторов библиотеки Mapnik. Например, штриховая линия на предыдущем изображении была создана при помощи следующего ниже фрагмента кода на Python:

```
stroke = mapnik.Stroke()
stroke.color = mapnik.Color("#008000")
stroke.width = 1.0
stroke.add_dash(5, 10)
symbolizer = mapnik.LineSymbolizer(stroke)
```

Как видите, чтобы задать внешний вид наносимой линии, символизатор линий LineSymbolizer использует штриховой объект Stroke. При использовании LineSymbolizer вы сначала создаете объект Stroke и устанавливаете различные варианты того, как вы хотите, чтобы строка была нарисована. Затем вы создаете свой экземпляр LineSymbolizer, передавая свой объект конструктору объекта LineSymbolizer:

```
symbolizer = mapnik.LineSymbolizer(stroke)
```

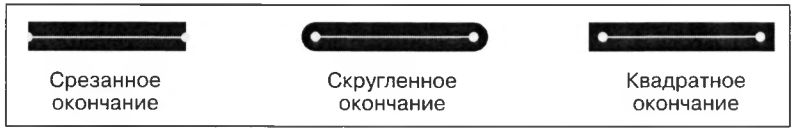
Настройка объекта `Stroke` выполняется следующими способами:

- `stroke.color = mapnik.Color("yellow")`: изменяет цвет наносимой линии. Объекты `Color` можно создавать, используя имя цвета, цветовой код HTML или отдельные значения R, G и B. Например:

```
color1 = mapnik.Color("yellow")
color2 = mapnik.Color("#f0f028")
color3 = mapnik.Color(240, 240, 40)
```

Помимо значений цвета RGB, можно воспользоваться дополнительным, четвертым значением для определения прозрачности цвета. Например, `mapnik.Color(240, 240, 40, 128)` создаст объект желтого цвета с 50%-ной прозрачностью.

- `stroke.width = 0.5`: установит ширину линии в пикселах. Отметим, что, для того чтобы провести линию, в библиотеке используется сглаживание, вследствие чего линия уже 1-го пиксела нередко выглядит лучше, чем линия целочисленной ширины;
- `stroke.opacity = 0.8`: установит степень прозрачности (транспарентности) линии. Прозрачность может колебаться от 0.0 (абсолютная прозрачность) до 1.0 (абсолютная непрозрачность);
- `stroke.line_cap = mapnik.line_cap.BUTT_CAP`: меняет вид окончаний наносимых линий. В библиотеке Mapnik поддерживаются три стандартных типа окончания линии (со срезанными окончаниями, со скругленными окончаниями и с квадратными окончаниями):



- `stroke.line_join = mapnik.line_join.MITER_JOIN`: настраивает вид «угла» линии, наносимого при изменении линией направления. Поддерживаются три стандартных стыка линий (стык без сглаживания, скругленный и со скосом):

- `stroke.add_dash(5, 7)`: позволяет добавлять к линии разрывы, чтобы она выглядела как штрих-пунктирная. Два передаваемых параметра составляют *длину штриха* и *длину разрыва*; линия будет проведена с заданной длиной штриха, и затем, прежде чем линия продолжится, будет вставлен разрыв заданной длины:

Отметим, что метод `add_dash()` можно вызвать несколько раз. Это позволяет вам создавать переменные последовательности точек или штрихов. Например:

```
stroke.add_dash(10, 2)
stroke.add_dash(2, 2)
stroke.add_dash(2, 2)
```

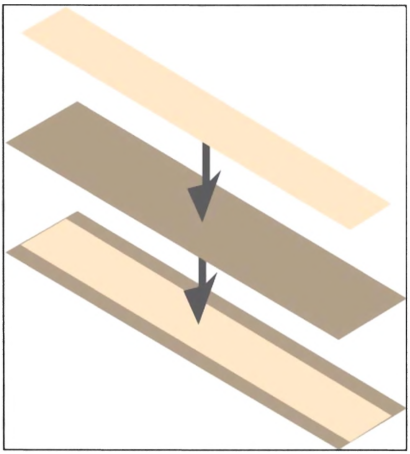
Эта последовательность вызовов приведет к следующему ниже повторяющемуся шаблону линии:

Единственное, что не сразу становится очевидным, когда вы начинаете работать с символизаторами, – это то, что для одного объекта можно использовать несколько символизаторов.

Например, рассмотрим следующий ниже фрагмент:

```
stroke = mapnik.Stroke()
stroke.color = mapnik.Color("#bf7a3a")
stroke.width = 7.0
roadEdgeSymbolizer = mapnik.LineSymbolizer(stroke)

stroke = mapnik.Stroke()
stroke.color = mapnik.Color("#ffd3a9")
stroke.width = 6.0
roadInteriorSymbolizer = mapnik.LineSymbolizer(stroke)

rule.symbols.append(roadEdgeSymbolizer)
rule.symbols.append(roadInteriorSymbolizer)
```

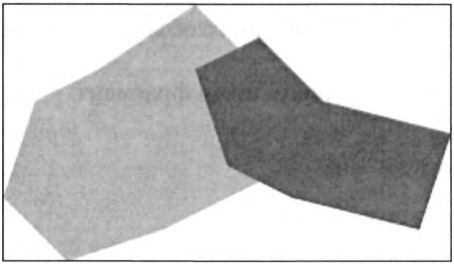
Два символизатора будут помещены один поверх другого, чтобы нанести ломаную линию, которая выглядит как дорога на карте города:



Этот прием можно использовать далеко не только для изображения дорог; творческое использование символизаторов – один из главных приемов, позволяющих достичь сложных визуальных эффектов при помощи библиотеки Mapnik.

Нанесение многоугольников

Символизатор многоугольников `PolygonSymbolizer` заполняет внутреннюю область многоугольника одним цветом:



Экземпляр символизатора многоугольников `PolygonSymbolizer` создается следующим образом:

```
symbolizer = mapnik.PolygonSymbolizer()
```

После создания экземпляра `PolygonSymbolizer` его можно сконфигурировать, установив следующие атрибуты:

- `symbolizer.fill = mapnik.Color("red")`: назначает цвет, используемый для заполнения внутренней области многоугольника;
- `symbolizer.opacity = 0.8`: изменяет уровень используемой прозрачности при заполнении внутренней области многоугольника;
- `symbolizer.gamma = 0.6`: при помощи значения `gamma` управляют величиной сглаживания, используемого для нанесения краев многоугольника. Если попро-

бовать нарисовать примыкающие многоугольники тем же цветом, то края, как правило, останутся видимыми; небольшое уменьшение значения gamma позволит убрать края примыкающих многоугольников, объединив их в один.

Нанесение текста

Символизатор текста TextSymbolizer наносит надписи на точечные, линейные и многоугольные объекты:

В своей основе символизатор текста TextSymbolizer используется довольно просто. Например, многоугольник на приведенной выше иллюстрации был промаркирован при помощи следующего ниже фрагмента кода:

```
symbolizer = mapnik.TextSymbolizer(mapnik.Expression("[label]"),
"DejaVu Sans Book", 10, mapnik.Color("black"))
```

Как видите, конструктор объекта Textsymbolizer принимает четыре отдельных параметра:

- объект mapnik.Expression, который задает выводимый текст. В этом примере выводимое значение будет взято из атрибута label источника данных;
- имя используемого для вывода текста шрифта. Чтобы увидеть, какие шрифты доступны, в командной строке Python наберите следующий ниже текст:

  ```
  import mapnik
  for font in mapnik.FontEngine.face_names():
      print(font)
  ```

  Кроме того, можно установить пользовательские шрифты, как описано на странице: https://github.com/mapnik/mapnik/wiki/UsingCustomFonts;
- размер шрифта в пикселах;
- объект mapnik.Color, используемый для вывода текста.

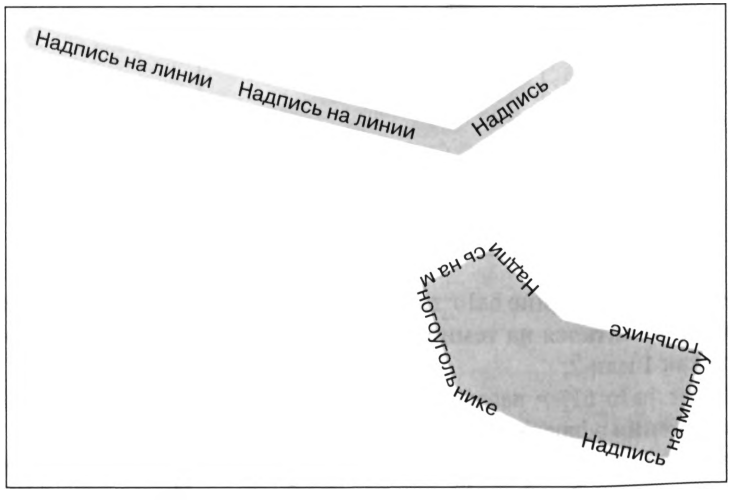
После того как это было создано, можно сконфигурировать объект `TextSymbolizer`, установив следующие атрибуты:

- `symbolizer.opacity = 0.5`: изменяет прозрачность, используемую при выводе текста;
- `symbolizer.label_placement = mapnik.label_placement.LINE_PLACEMENT`: атрибут `label_placement` управляет позиционированием текста. Значение `POINT_PLACEMENT` (по умолчанию) просто рисует текст в центре геометрии, как показано здесь:

Если его поменять на `LINE_PLACEMENT`, то надпись будет выведена вдоль ломаной или вдоль периметра многоугольника:

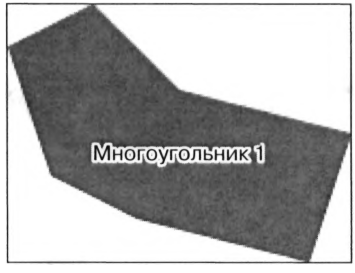
Отметим, что если использовать `LINE_PLACEMENT` для геометрии точки, то текст не выводится совсем, так как внутри точки нет линий;

- `symbolizer.label_spacing = 30`: по умолчанию надпись для каждого объекта выводится всего один раз. Установив `label_spacing` в заданное число пикселов, надпись будет повторена, как показано ниже:



- `symbolizer.allow_overlap = True`: по умолчанию библиотека Mapnik избегает наложения двух надписей друг на друга путем незначительного смещения надписей либо путем сокрытия одной из них. Установка `allow_overlap` в `True` отменяет это поведение, разрешая изображение нескольких надписей поверх друг друга:

- `symbolizer.halo_radius = 1`: если надпись наносится с темным текстом на темном фоне, то такую надпись обычно очень трудно читать. Можно воспользоваться этим атрибутом, чтобы нанести вокруг надписи эффект «ореола», который делает темную надпись по-прежнему видимой на темном фоне. Например:



  Отметим, что значение `halo_radius` указывается в пикселах; чтобы текст гарантируемо читался на темном фоне, обычно достаточно малого значения, такого как 1 или 2;
- `symbolizer.halo_fill = mapnik.Color("white")`: назначает цвет, используемый для нанесения эффекта ореола вокруг надписи.

На самом деле вариантов управления отображением надписей при помощи символизатора текста `TextSymbolizer` гораздо больше. Например, можно управлять разбиением надписи на несколько строк, определять межсимвольный и межстрочный интервалы и даже динамически выбирать атрибут отображения, исходя из длины текста и свободного места. Для получения дополнительной информации о доступных параметрах, пожалуйста, проконсультируйтесь с документацией по библиотеке Mapnik.

В дополнение к символизатру текста `TextSymbolizer` еще имеется так называемый символизатор знаков `ShieldSymbolizer`, который совмещает объект `TextSymbolizer` с объектом `PointSymbolizer`, позволяя в заданной точке выводить изображение и надпись. Из-за его сложности `ShieldSymbolizer` в этой книге рассматриваться не будет; если вы захотите изучить этот символизатор, то обнаружите, что в нем используются те же самые атрибуты, что и в символизаторах `TextSymbolizer` и `PointSymbolizer`.

Вывод растровых изображений

При помощи растровых изображений часто выводят базовую карту (подложку), на которую наносятся ваши векторные данные. Чтобы нарисовать растровое изображение, обычно применяется символизатор растра `RasterSymbolizer`. Например:

```
symbolizer = mapnik.RasterSymbolizer()
```

Символизатор растра `RasterSymbolizer` выводит на карту содержимое растрового источника данных слоя. Наряду с тем, что существует несколько расширенных настроек управления выводом растрового изображения, в большинстве случаев

атрибут прозрачности opacity будет единственной опцией, которая вас, скорее всего, заинтересует. Как и следует ожидать, он устанавливает прозрачность изображения, позволяя выводить несколько полупрозрачных изображений, один поверх другого.

Карты и слои

После настройки своих источников данных, символизаторов, правил и стилей можно объединить их в слои библиотеки Mapnik и поместить все слои вместе на карту. Для этого вы сначала создаете объект mapnik.Map, который будет представлять карту в целом:

```
map = mapnik.Map(width, height, srs)
```

Вы передаете ширину и высоту изображения цифровой карты, которую вы хотите сгенерировать, в пикселах, и в параметре srs – необязательную строку инициализации в формате PROJ.4. Если вы не указываете систему пространственной привязки, то в карте будет использованы параметры +proj=latlong +datum=WGS84, то есть географические координаты широты и долготы по датуму WGS84.

После создания карты вы выбираете цвет ее фона и затем, вызывая метод map.append_style(), добавляете в карту ваши всевозможные стили:

```
map.background_color = mapnik.Color('white')
map.append_style("countryStyle", countryStyle)
map.append_style("roadStyle", roadStyle)
map.append_style("pointStyle", pointStyle)
map.append_style("rasterStyle", rasterStyle)
...
```

Затем внутри карты вам нужно создать всевозможные слои. Для этого вы создаете объект mapnik.Layer, который представляет каждый слой карты:

```
layer = mapnik.Layer(layerName, srs)
```

Каждому слою назначается уникальное имя. Помимо него, слой может иметь необязательную ассоциированную с ним пространственную привязку. Строковый параметр srs опять же представляет собой строку инициализации в формате PROJ.4; если пространственная привязка не задана, то в слое будет использовано значение +proj=latlong +datum=WGS84.

После создания слоя карты вы присваиваете ему источник данных и выбираете связанный с этим слоем стиль(и), при этом каждый стиль идентифицируется по имени:

```
layer.datasource = myDatasource
layer.styles.append("countryStyle")
layer.styles.append("rasterStyle")
...
```

В заключение вы добавляете новый слой к карте:

```
map.layers.append(layer)
```

При помощи слоев карты можно в одном изображении совмещать несколько источников данных. При помощи них можно также выборочно отображать или скрывать информацию в зависимости от текущего масштабного уровня. Например, рассмотрим две нижеследующие карты:

Очевидно, рисовать улицы на карте всего мира нет никакого смысла. То же самое и с контуром страны на карте мира – ее контур имеет слишком крупный масштаб, чтобы детализированно отображать побережье отдельного города. Но если ваше приложение позволит пользователю менять уровень приближения, начиная с карты мира вплоть до отдельной улицы, то вам придется использовать единый набор стилей Mapnik, генерирующий карту независимо от масштаба, в котором вы ее изображаете.

Самый легкий способ сделать это – установить для слоя минимальный и максимальный масштабные коэффициенты. Например:

```
layer.minzoom = 1/100000
layer.maxzoom = 1/200000
```

Слой будет показан, только когда текущий масштабный коэффициент карты будет находиться в пределах этого диапазона. Это целесообразно делать, когда ваш источник данных должен использоваться для отображения карты только в определенном масштабе, например используя данные береговых линий высокого разрешения, только когда пользователь увеличивает масштаб.

Визуализация карты

После создания вашего объекта `mapnik.Map` и настройки внутри него различных символизаторов, правил, стилей, источников данных и слоев вы, наконец, готовы преобразовать свою карту в графическое изображение.

Прежде чем выполнить визуализацию карты, убедитесь, что вы настроили для карты соответствующую ограничительную рамку, с тем чтобы карта показывала именно тот регион мира, которым вы интересуетесь. Вы можете сделать это путем вызова метода `zoom_to_box()`, чтобы явным образом установить ограничительную рамку карты в заданный набор координат, либо путем вызова метода `map.zoom_all()`, чтобы границы карты были установлены автоматически, исходя из отображаемых данных.

Установив ограничительную рамку, можно сгенерировать изображение карты, вызвав для этого функцию render_to_file(), как показано ниже:

```
mapnik.render_to_file(map, 'map.png')
```

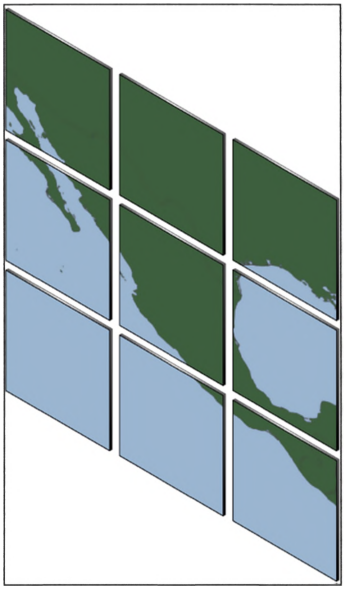
Параметрами являются объект mapnik.Map и имя графического файла, куда карта будет записана. Если вам нужен больший контроль над форматом генерируемого изображения, то можно добавить дополнительный параметр, в котором указывается графический формат, как показано ниже:

```
mapnik.render_to_file(map, 'map.png', 'png256')
```

Поддерживаются следующие графические форматы:

Графический формат	Описание
.png	32-разрядное изображение в формате PNG
.png256	8-разрядное изображение в формате PNG
.jpeg	Изображение в формате JPEG
.svg	Изображение в формате SVG
.pdf	Файл PDF
.ps	Файл в формате PostScript

Функция render_to_file() работает хорошо, когда вы хотите сгенерировать из всей своей карты единственное изображение. Другой полезный способ визуализации карты состоит в генерировании нескольких четырехугольных сегментов, которые затем могут быть склеены, чтобы вывести карту в более высоком разрешении:



Библиотека Mapnik предлагает полезную функцию для создания четырехугольных сегментов из одной карты, как показано ниже:

```
mapnik.render_tile_to_file(map, xOffset, yOffset,
                           width, height, fileName, format)
```

Параметры этой функции следующие:

- `map` – это объект `mapnik.Map`, содержащий данные географической карты;
- `xOffset` и `yOffset` задают верхний левый угол сегмента в прямоугольных координатах;
- `width` и `height` задают размер сегмента в прямоугольных координатах;
- `fileName` – имя файла, куда будет сохранен сегмент изображения карты;
- `format` – формат файла, в котором будет сохранено данное изображение.

Эту функцию можно вызывать неоднократно для создания отдельных сегментов. Например:

```
for x in range(NUM_TILES_ACROSS):
    for y in range(NUM_TILES_DOWN):
        xOffset = TILE_SIZE * x
        yOffset = TILE_SIZE * y
        tileName = "tile_{}_{}.png".format(x, y)
        mapnik.render_tile_to_file(map, xOffset, yOffset,
                                   TILE_SIZE, TILE_SIZE,
                                   tileName, "png")
```

Заключение

В этой главе мы подробно изучили инструмент визуализации цифровых карт – библиотеку Python `Mapnik`. Мы узнали, что карты состоят из нескольких слоев, наносимых один поверх другого при помощи алгоритма живописца. Мы увидели, что каждый слой имеет источник данных и список стилей, которые определяют, каким образом данные должны быть нанесены. К стилям обращаются по имени, и поэтому они могут совместно использоваться различными слоями карты.

Мы узнали, что у каждого стиля есть связанный с ним список правил, а у каждого правила может дополнительно иметься фильтр, ограничивающий набор отображаемых геообъектов, а также список символизаторов, которые управляют тем, как соответствующие геообъекты должны отображаться.

Затем мы увидели, как объект `mapnik.Map` совмещает стили и слои карты и как при помощи карты можно увеличивать масштаб конкретного региона мира перед выполнением визуализации на основе содержимого карты.

Мы собрали все эти понятия в простую компьютерную картографическую программу, особо не углубляясь в тонкости библиотеки Mapnik. Мы изучили четыре главных типа источников данных: `Shapefile`, `PostGIS`, `Gdal` и `MemoryDatasource`. Мы увидели, каким образом правила, фильтры и стили взаимодействуют между собой, и исследовали главные типы символизаторов, которые вы, скорее всего, будете использовать в своих программах: символизатор точек `PointSymbolizer` для

отрисовки геометрий точек, символизатор линий `LineSymbolizer` для отрисовки линейных объектов и контура многоугольника, символизатор многоугольников `PolygonSymbolizer` для заполнения внутренней области многоугольника, символизатор текста `TextSymbolizer` для нанесения надписей и символизатор растра `RasterSymbolizer` для размещения на карте растровых изображений.

Затем на примере использования масштабных коэффициентов для выборочного отображения или сокрытия слоев карты мы рассмотрели способы взаимодействия карты и расположенных в слое объектов между собой и завершили обзором различных методов визуализации карт.

В следующей главе мы объединим пространственные базы данных с библиотекой Mapnik, чтобы создать технологичное геопространственное приложение под названием DISTAL.

Глава 8

Работа с пространственными данными

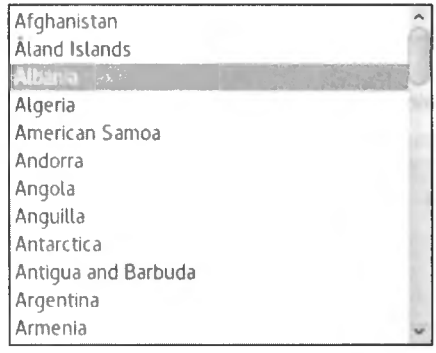
В этой главе мы применим и разовьем знания, полученные в предыдущих главах, создав технологичное веб-приложение, выполняющее идентификацию береговых линий, городов и озер в заданном радиусе, под названием **DISTAL**, англ. название Distance-based Identification of Shorelines, Towns, and Lakes, аббревиатура переводится как «периферия». В процессе создания этого приложения мы научимся:

- работать со значительными объемами геоданных, хранящихся в базе данных PostGIS;
- выполнять сложные пространственные запросы к базе данных;
- делать точные пространственные запросы на основе расстояний;
- интегрировать библиотеку Mapnik в веб-приложение для отображения результатов на карте.

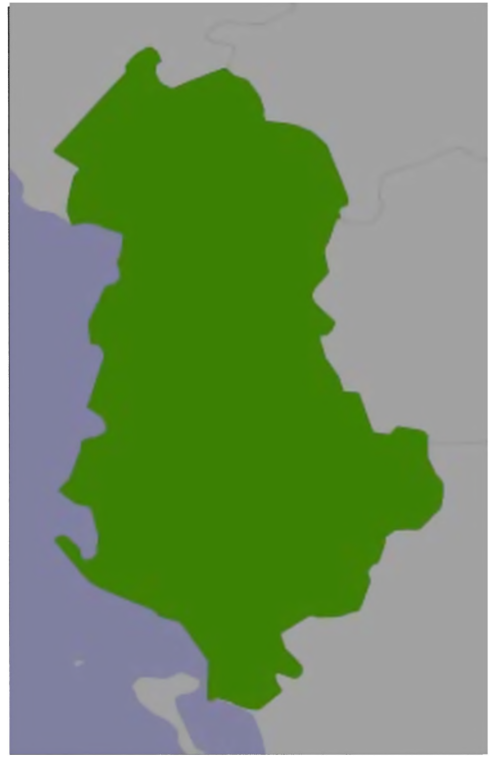
Описание приложения DISTAL

Приложение DISTAL будет иметь следующую упрощенную последовательность операций:

1. Пользователь начинает с выбора страны, с которой он желает работать:

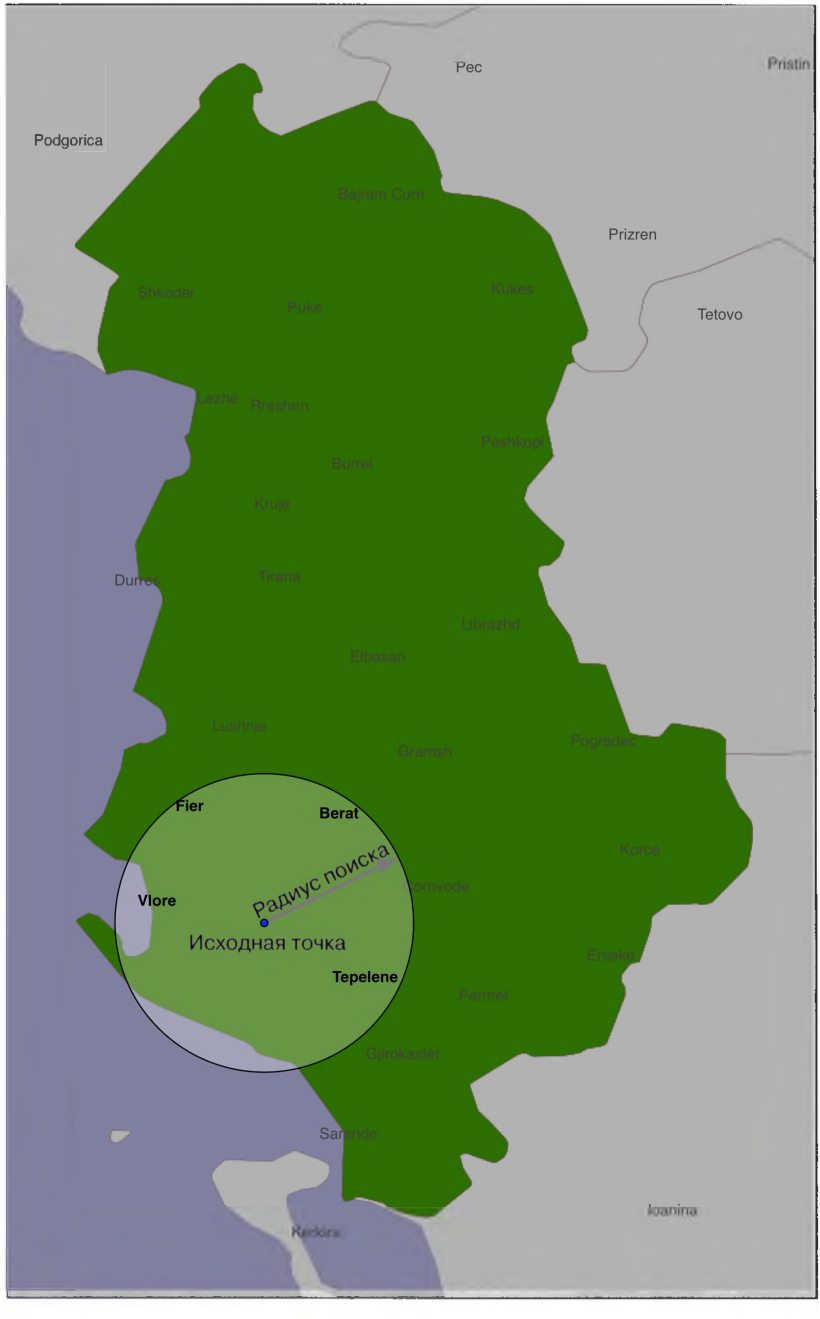


2. Затем выводится простая карта страны:

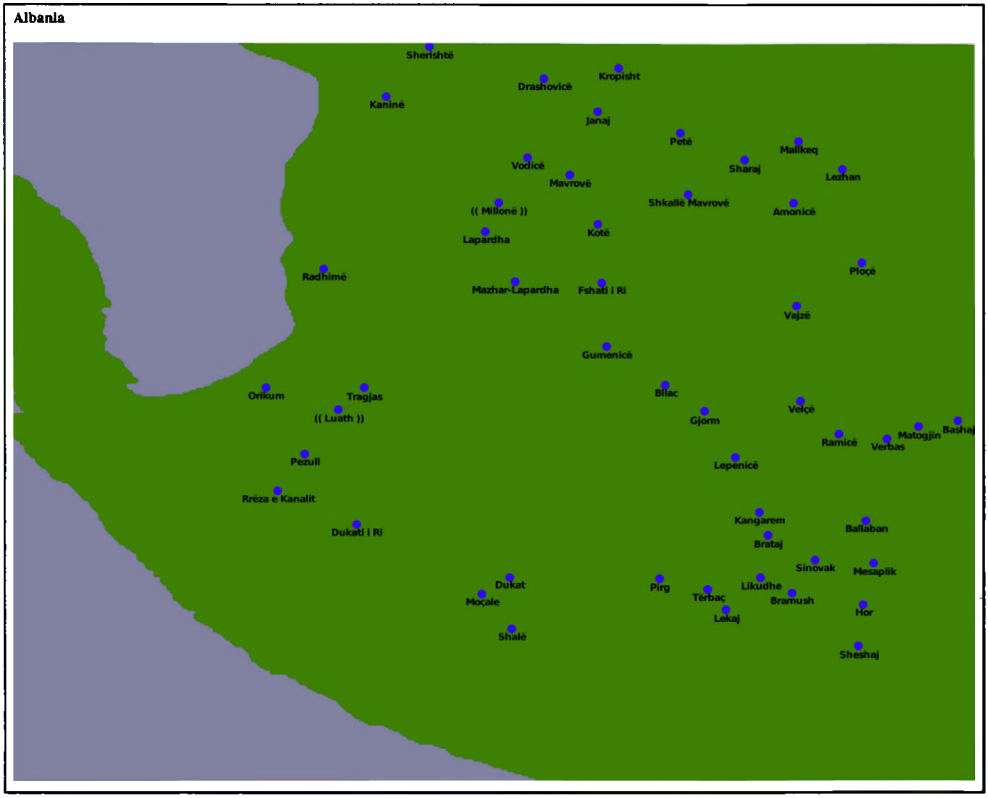


3. Пользователь выбирает нужный радиус в милях и нажимает на любую точку внутри страны:

4. Система определяет все крупные и малые города в пределах заданного радиуса от этой точки:



5. В заключение все полученные геообъекты выводятся пользователю в более высоком разрешении для просмотра или печати:



Мы реализуем систему DISTAL как серию сценариев CGI. Будучи предельно упрощенной и испытывающей нехватку безукоризненного внешнего вида и манер настоящей «скользящей карты»[1], как в картографической веб-службе Карты Google, эта последовательность позволяет концентрироваться на базовых геопространственных аспектах данного приложения, не увязая в сложности создания полнофункционального веб-приложения.

Проектирование и конструирование базы данных

Начнем проектировать наше приложение DISTAL с анализа различных элементов данных, которые для него потребуются:

---

[1] Скользящая, или подвижная, карта (англ. slippy map) – современная веб-карта с функционалом масштабирования и панорамирования (карта «скользит», когда вы двигаете мышью). Также веб-интерфейс для просмотра преобразованных в карту геоданных таким образом. – *Прим. перев.*

- список всех стран. Каждая страна должна содержать простую карту пограничной полосы, которая может выводиться пользователю;
- подробные границы озер и береговых линий всего мира. Список всех основных крупных и малых городов мира;
- по каждому крупному/малому городу потребуются его наименование и географическое положение.

К счастью, эти данные легкодоступны[1]:

- список стран и их контуров включен в набор данных границ стран мира World Borders Dataset;
- границы озер и береговых линий (а также другие границы водораздела, такие как острова на озерах) легкодоступны при помощи географической базы данных береговых линий GSHHG;
- данные по крупным и малым городам можно найти в двух местах: база данных GNIS предлагает официальные топонимические данные по США, тогда как база данных сервера географических названий GEOnet обеспечивает подобные данные по остальной части мира.

Рассматривая эти источники данных, можно приступать к разработке схемы базы данных для системы DISTAL, которая будет выглядеть следующим образом:

Поле `level` (уровень) в таблице береговых линий `shorelines` соответствует значению уровня в базе данных GSHHG: значение 1 обозначает береговую линию, 2 – озеро, 3 – остров на озере и 4 – водоем на острове, расположенном на озере.

Несмотря на то что эта схема выглядит очень простой, для начала ее будет вполне достаточно. Воспользуемся этой схемой, чтобы создать нашу пространственную базу данных и выполнить настройку схемы базы данных. Настройка состоит из нижеследующих этапов.

1. Откройте окно терминала или командной строки и наберите следующую команду SQL:

```
createdb distal
```

Напомним, что следует добавить опцию `-U postgres` либо воспользоваться утилитой `sudo` для этой команды и всех последующих, если необходимо запустить СУБД Postgres под другой учетной записью пользователя.

[1] Файлы всех используемых в книге геопространственных данных имеются в составе прилагаемых к книге примеров программ. – *Прим. перев.*

2. Создайте нового пользователя, чтобы получить доступ к этой базе данных, введя следующую команду SQL:

   ```
   createuser -P distal_user
   ```

   Будет предложено ввести пароль для этого нового пользователя. Убедитесь в правильности вводимого вами пароля.
3. Запустите интерфейс командной строки СУБД Postgres, набрав следующее:

   ```
   psql distal
   ```

4. Разрешите пользователю distal_user получать доступ к базе данных distal, введя следующее:

   ```
   GRANT ALL PRIVILEGES ON DATABASE distal TO distal_user;
   ```

5. Пространственно активируйте новую базу данных, набрав следующую команду:

   ```
   CREATE EXTENSION postgis;
   ```

Теперь можно выйти из клиента командной строки СУБД Postgres, введя \q и нажав *Return*. Далее мы напишем сценарий Python, который создаст наши таблицы базы данных. Впрочем, перед этим нам придется где-то сохранить исходный код системы DISTAL. Создайте новый каталог с именем DISTAL в удобном для вас месте. В этом каталоге создайте новый сценарий Python под названием create_db.py и наберите туда следующее:

```
import psycopg2
connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()

cursor.execute("DROP TABLE IF EXISTS countries")
cursor.execute("""
  CREATE TABLE countries (
    id SERIAL,
    name VARCHAR(255),
    outline GEOMETRY(GEOMETRY, 4326),
    PRIMARY KEY (id))
""")
cursor.execute("""
  CREATE INDEX countryIndex ON countries
  USING GIST(outline)
""")

cursor.execute("DROP TABLE IF EXISTS shorelines")
cursor.execute("""
  CREATE TABLE shorelines (
    id SERIAL,
    level INTEGER,
```

```
    outline GEOMETRY(GEOMETRY, 4326),
    PRIMARY KEY (id))
""")

cursor.execute("""
  CREATE INDEX shorelineIndex ON shorelines
    USING GIST(outline)
""")

cursor.execute("DROP TABLE IF EXISTS places")
cursor.execute("""
  CREATE TABLE places (
    id SERIAL,
    name VARCHAR(255),
    position GEOGRAPHY(POINT, 4326),
    PRIMARY KEY (id))
""")

cursor.execute("""
  CREATE INDEX placeIndex ON places
    USING GIST(position)
""")

connection.commit()
```

В этой главе очень много исходного кода, но от вас не требуется набирать все это вручную. Вы можете скачать полную копию приложения DISTAL в составе примеров программного кода, которые прилагаются к этой главе.

Отметим, что мы определяем контуры с типом столбца `GEOMETRY(GEOMETRY, 4326)`. Первое слово `GEOMETRY` обозначает тип столбца; вместо географических мы используем геометрические столбцы, то есть с типом geometry, чтобы избежать проблем с вычислением ограничительных рамок и других значений, которые, к сожалению, не работают со столбцами geography. Второе слово `GEOMETRY` внутри круглых скобок сообщает расширению PostGIS, что мы можем употребить значение с любым типом геометрии, которое потребуется; это позволяет нам хранить в качестве контура многоугольник или составной многоугольник, в зависимости от типа контура, который нам придется хранить. Мы также последуем рекомендациям и определим для наших данных значения идентификатора пространственной привязки (SRID). Кроме того, для этих данных мы воспользуемся датумом WGS84 и географическими координатами широты и долготы, которые соответствуют значению SRID 4326.

В таблице `places` с топонимикой мы используем `GEOGRAPHY(POINT, 4326)`, то есть столбец `position` с типом `GEOGRAPHY`, чтобы внутри базы данных можно было выполнять точные запросы на основе расстояний. На этот раз содержимое топонима ограничено значением единственной точки, и для этих данных мы снова используем значение SRID 4326.

Пойдем дальше и выполним программу `create_db.py`; она должна выполниться безошибочно, должным образом настроив для вас схему базы данных.

Скачивание и импорт данных

Как описано в предыдущем разделе, в приложении DISTAL используются четыре отдельных общедоступных набора геоданных:

- набор данных границ стран мира World Borders Dataset;
- географическая база данных береговых линий GSHHG с высоким разрешением;
- база данных географических названий GNIS по США;
- список неамериканских топонимов базы данных GEONet.

За дополнительной информацией по поводу этих источников данных обратитесь к *главе 4 «Источники геоданных»*.

Разберем по очереди процессы скачивания и импортирования каждого из этих наборов данных.

Набор данных границ стран мира

Ранее вы уже скачали копию этого набора данных. Создайте подкаталог `data` внутри каталога `DISTAL` и поместите в него копию каталога `TM_WORLD_BORDERS-0.3`. Затем внутри каталога `DISTAL` создайте новую программу Python под названием `import_world_borders.py` и введите туда следующую ниже программу:

```
import os.path
import psycopg2
import osgeo.ogr

connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()

cursor.execute("DELETE FROM countries")

srcFile = os.path.join("data", "TM_WORLD_BORDERS-0.3",
                               "TM_WORLD_BORDERS-0.3.shp")
shapefile = osgeo.ogr.Open(srcFile)
layer = shapefile.GetLayer(0)
num_done = 0

for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    name = feature.GetField("NAME")
    wkt = feature.GetGeometryRef().ExportToWkt()
    cursor.execute("INSERT INTO countries (name,outline) " +
                   "VALUES (%s, ST_GeometryFromText(%s, 4326))",
                   (name, wkt))
    num_done = num_done + 1

connection.commit()

print("Импортировано {} стран".format(num_done))
```

В этой программе довольно легко разобраться; мы просто используем библиотеку OGR, которая загружает содержимое файла фигур, и адаптер psycopg2 для Python, который скопирует данные в таблицу стран countries. При выполнении этой программы содержимое набора данных границ стран мира будет импортировано в вашу базу данных.

Географическая база данных береговых линий GSHHG

Скачайте копию географической базы данных GSHHG (http://www.ngdc.noaa.gov/mgg/shorelines/gshhs.html), в случае если вы еще этого не сделали. Удостоверьтесь, что вы скачиваете полную базу данных в формате файла фигур. Поместите получившийся каталог GSHHS_shp в свой подкаталог данных data.

Напомним, что база данных GSHHG раньше называлась GSHHS, и именно по этой причине в нескольких местах, таких как имя этого файла фигур, все еще встречается сокращение GSHHS.

Географическая база данных береговых линий GSHHG состоит из отдельных файлов фигур, в которых определены границы водораздела в пяти различных разрешениях. Для приложения DISTAL нам нужно импортировать четыре из них (береговая линия, озеро, остров на озере и водоем на острове, расположенном на озере) в полном разрешении. Этот файл фигур хранится в подкаталоге с именем f, которое обозначает полное (full) разрешение.

Чтобы импортировать эти данные в базу данных DISTAL, создайте новую программу под названием import_gshhg.py внутри вашего каталога DISTAL и наберите туда следующую ниже программу на Python:

```
import os.path
import psycopg2
import osgeo.ogr

connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()

cursor.execute("DELETE FROM shorelines")

for level in [1, 2, 3, 4]:
    num_done = 0
    srcFile = os.path.join("data", "GSHHS_shp", "f",
                           "GSHHS_f_L{}.shp".format(level))
    shapefile = osgeo.ogr.Open(srcFile)
    layer = shapefile.GetLayer(0)

    for i in range(layer.GetFeatureCount()):
        feature = layer.GetFeature(i)
        geometry = feature.GetGeometryRef()
        if geometry.IsValid():
            cursor.execute("INSERT INTO shorelines " +
```

```
                    "(level,outline) VALUES " +
                    "(%s, ST_GeometryFromText(%s, 4326))",
                    (level, geometry.ExportToWkt()))
        num_done = num_done + 1

    connection.commit()

    print("Импортировано {} береговых линий на уровне
          {}".format(num_done, level))
```

Напомним о необходимости ввести пароль, который вы установили для своего пользователя `distal_user` **при вызове функции** `psycopg2.connect()`.

Отметим, что, прежде чем вставить геометрию в базу данных, мы должны проверить ее корректность. Это объясняется небольшими проблемами, связанными с несколькими геометриями в базе данных GSHHG. Теоретически можно было бы эти ошибочные записи исправить, но поскольку это довольно сложно выполнить, мы их просто пропустим.

Когда вы запустите эту программу, то на ее выполнение потребуется несколько минут, поскольку будет импортировано более 180 000 многоугольников с высоким разрешением.

Географические названия США

Список географических названий США можно скачать со страницы http://geonames.usgs.gov/domestic. Щелкните по ссылке **Download Domestic Names** (Скачать внутренние географические названия) и выберите опцию **Download all national features in one .zip file** (Скачать все национальные геообъекты одним файлом .zip); это гиперссылка с надписью NationalFile. В результате вы получите файл объемом свыше 80 Мб в сжатом виде с именем `NationalFile_YYYYMMDD.zip`, где `YYYYMMDD` – это метка даты, обозначающая время обновления файла. И снова распакуйте полученный архив `.zip` и переместите текстовый файл `NationalFile_YYYYMMDD.txt` в ваш подкаталог `data`.

Файл, который вы только что скачали, представляет собой файл с разделением полей данных символом вертикальной черты (`|`), где каждый столбец обозначается, как показано ниже:

```
FEATURE_ID|FEATURE_NAME|FEATURE_CLASS|...|DATE_EDITED
399|Agua Sal Creek|Stream|AZ|...|02/08/1980
400|Agua Sal Wash|Valley|AZ|...|02/08/1980
```

Первая строка содержит имена различных полей. Несмотря на то что в файле полей много, для нас особый интерес вызывают всего четыре из них[1]:

- поле `FEATURE_NAME` (Имя геообъекта) содержит название географического положения в символьной кодировке UTF-8;

[1] Описание полей приведено на странице http://geonames.usgs.gov/apex/f?p=136:8. – *Прим. перев.*

- поле FEATURE_CLASS (Класс геообъекта) сообщает нам, с каким геообъектом мы имеем дело, таким как Stream (Поток) или Valley (Долина). Для приложения DISTAL многие геообъекты не понадобятся, например названия заливов, пляжей, мостов и месторождений нефти. Фактически есть только один интересующий нас класс геообъектов: Populated Place, то есть названия населенных пунктов, таких как большие и малые города, деревни и пр.;
- поля PRIM_LONG_DEC и PRIM_LAT_DEC содержат долготу и широту географического положения в десятичных градусах. Согласно документации, эти координаты используют датум NAD83 вместо датума WGS84 в остальных импортируемых нами данных.

Приступим к импорту данных из этого файла в нашу таблицу топонимов places базы данных. Так как значения широты и долготы основаны на датуме NAD83, мы возьмем библиотеку pyproj, которая для совместимости координат с остальными используемыми нами данными будет конвертировать их в датум WGS84.

Строго говоря, этот этап не особо нужен. Для перевода координат из NAD83 в WGS84 мы используем библиотеку pyproj. Однако, учитывая, что все импортируемые данные лежат в пределах США и эти два датума для таких точек идентичны, библиотека pyproj на самом деле вообще не изменит координат. Но мы все равно проделаем это, следуя рекомендуемой практике, связанной с получением для наших данных пространственной привязки и по мере необходимости ее трансформацией, даже если эта трансформация временами выполняется вхолостую.

Напишем программу импорта этих данных в базу данных. В вашем главном каталоге DISTAL создайте новый файл с именем import_gnis.py и введите туда следующую ниже программу на Python:

```
import psycopg2
import os.path
import pyproj

connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()

cursor.execute("DELETE FROM places")

srcProj = pyproj.Proj(proj='longlat', ellps='GRS80',
                      datum='NAD83')
dstProj = pyproj.Proj(proj='longlat', ellps='WGS84',
                      datum='WGS84')

f = file(os.path.join("data",
                      "NationalFile_YYYYMMDD.txt"), "r",
                      encoding='UTF8')
heading = f.readline() # Проигнорировать имена полей.

num_inserted = 0
```

```
for line in f:
    parts = line.rstrip().split("|")
    featureName = parts[1]
    featureClass = parts[2]
    lat = float(parts[9])
    long = float(parts[10])

    if featureClass == "Populated Place":
        long,lat = pyproj.transform(srcProj, dstProj,
                                    long, lat)
        cursor.execute("INSERT INTO places " +
                       "(name, position) VALUES (%s, " +
                       "ST_SetSRID(" +
                       "ST_MakePoint(%s,%s), 4326))",
                       (featureName, long, lat))

        num_inserted += 1
        if num_inserted % 1000 == 0:
            connection.commit()

connection.commit()
f.close()

print("Вставлено {} топонимов".format(num_inserted))
```

Проверьте, что в вызове функции `psycopg2.connect()` **вы указали пароль для** `distal_user`. **Также убедитесь, что для скачанного вами файла** `NationalFile_YYYYMMDD.txt` **вы используете корректное имя в полном соответствии с его меткой даты.**

Отметим, что мы фиксируем изменения в базе данных для каждой 1000 записей. Это ускоряет процесс импорта. Мы также используем функцию `ST_SetSRID()` для назначения SRID геометриям точки перед их вставкой – это нужно сделать в связи с тем, что мы поручили PostGIS принимать только те геометрии, у которых есть значение SRID `4326`.

После запуска программы ее выполнение займет несколько минут. По ее завершении в таблицу `places` будет импортировано более 200 000 географических названий США.

Географические названия остальных мест

Скачайте список неамериканских географических названий со страницы http://geonames.nga.mil/gns/html/namefiles.html, в случае если вы еще этого не сделали, нажав для этого на гиперссылку **Entire country files dataset** (Весь набор страновых файлов). Отметим, что скачиваемый файл имеет большой размер – более 460 Мб в сжатом виде, поэтому на его скачивание может потребоваться продолжительное время.

После того как файл закачан, у вас получится файл с именем `geonames_YYYYMMDD.zip`, где `YYYMMDD` – это, напомним, метка даты обновления файла. Распакуйте файл, в результате чего будет создан каталог, содержащий два текстовых файла:

Countries.txt и Countries_disclaimer.txt. Нам нужен файл с именем Countries.txt, поэтому скопируйте его в свой подкаталог data.

Файл Countries.txt слишком большой и, скорее всего, в текстовом редакторе не откроется. Если бы можно было его просмотреть, то вы бы увидели, что он представляет собой файл с разделением полей данных вертикальной чертой (|) в символьной кодировке UTF-8. Первые несколько строк этого файла выглядят примерно так:

```
RC UFI ... FULL_NAME_ND_RG NOTE MODIFY_DATE
1 -1307834 ... Pavia 1993-12-21
1 -1307889 ... Santa Anna gjgscript 1993-12-21
```

Как и в случае с данными географических названий США, здесь имеется гораздо больше геообъектов, чем нам нужно для приложения DISTAL. Поскольку нас интересуют только официальные названия больших и малых городов, нам следует отфильтровать эти данные следующим образом:

- поле **классификации геообъекта** FC (Feature Classification) сообщает, с каким типом геообъектов мы имеем дело. Нам нужны геообъекты, чье значение FC равно «P» (populated place), то есть населенный пункт;
- поле с **типом имени** NT (Name Type) дает информацию о статусе названия этого геообъекта. Нам нужны названия, чье значение NT равно «N» (approved name), то есть согласованное (официальное) название;
- поле с **кодом обозначения геообъекта** DSG (feature designation code) дает более подробную, чем поле FC, информацию о типе геообъекта. В рамках приложения DISTAL нас интересуют геообъекты, чье значение DSG равно «PPL» (populated place), то есть населенный пункт, «PPLA» (administrative capital), то есть административный центр, или «PPLC» (capital city), то есть столица.

Если вы хотите увидеть полный список всех кодов обозначения геообъектов, то перейдите на http://geonames.nga.mil/namesgaz и нажмите на ссылку **Designation Codes** (Коды обозначения) в боковой панели слева под **Lookup Tables** (Таблицы поиска). В результате будет выведена страница, где можно выполнить поиск конкретного кода обозначения геообъекта; если вы оставите все поля незаполненными и щелкнете по кнопке **Search Designation Codes** (Искать коды обозначения), то появится окно с самыми разными кодами обозначения геообъектов и объяснением их значений.

По каждому геообъекту также имеется несколько различных версий географических названий; нам нужно полное название в нормальном порядке чтения, которое находится в поле с именем FULL_NAME_RO.

В нем находится все, что нам нужно для импорта в таблицу places нашей базы данных географических названий, находящихся за пределами США. В каталоге DISTAL создайте новый сценарий Python под названием import_geonames.py и наберите туда следующую ниже программу:

```
import psycopg2
import os.path
```

```
connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()

f = open(os.path.join("data",
                      "Countries.txt"), "r",
                      encoding='UTF8')

heading = f.readline() # Игнорировать имена полей.

num_inserted = 0

for line in f:
    parts = line.rstrip().split("\t")
    lat = float(parts[3])
    long = float(parts[4])
    featureClass = parts[9]
    featureDesignation = parts[10]
    nameType = parts[17]
    featureName = parts[22]

    if (featureClass == "P" and nameType == "N" and
        featureDesignation in ["PPL", "PPLA", "PPLC"]):
        cursor.execute("INSERT INTO places " +
                       "(name, position) VALUES (%s, " +
                       "ST_SetSRID(" +
                       "ST_MakePoint(%s,%s), 4326))",
                       (featureName, long, lat))

        num_inserted = num_inserted + 1
        if num_inserted % 1000 == 0:
            connection.commit()

f.close()
connection.commit()

print("Вставлено {} топонимов".format(num_inserted))
```

Отметим, что мы не удаляем существующее содержимое таблицы places. Это объясняется тем, что мы уже импортировали в эту таблицу немного данных. Если вам нужно импортировать данные повторно, то необходимо по очереди выполнить сначала сценарий import_gnis.py и затем сценарий import_geonames.py.

Пойдем дальше и выполним эту программу. Ее работа займет приблизительно 10–15 минут в зависимости от быстродействия вашего компьютера – так как вы добавляете в базу данных более 3 миллионов географических названий.

Реализация приложения DISTAL

Располагая данными, теперь можно приступить к реализации непосредственно самого приложения DISTAL. Чтобы ничего не усложнять, мы реализуем пользовательский интерфейс на основе сценариев CGI.

Что такое сценарий CGI?

Хотя подробности написания сценариев CGI выходят за рамки этой книги, все же отметим, что, в сущности, идея состоит в том, чтобы печатать исходные выходные данные HTML в стандартный поток `stdout` и обрабатывать приходящие из браузера входные параметры CGI, используя встроенный модуль `cgi`.

Чтобы выполнить программу на Python как сценарий CGI в Mac OS X или Linux, необходимо сделать две вещи. Во-первых, следует в начало сценария добавить строку «шебанг», как показано ниже:

```
#!/usr/bin/env python3
```

Точный путь, который вы используете, будет зависеть от того, где на вашем компьютере установлен Python. Во-вторых, необходимо сделать сценарий исполнимым, как показано ниже, предоставив пользователям право на его выполнение:

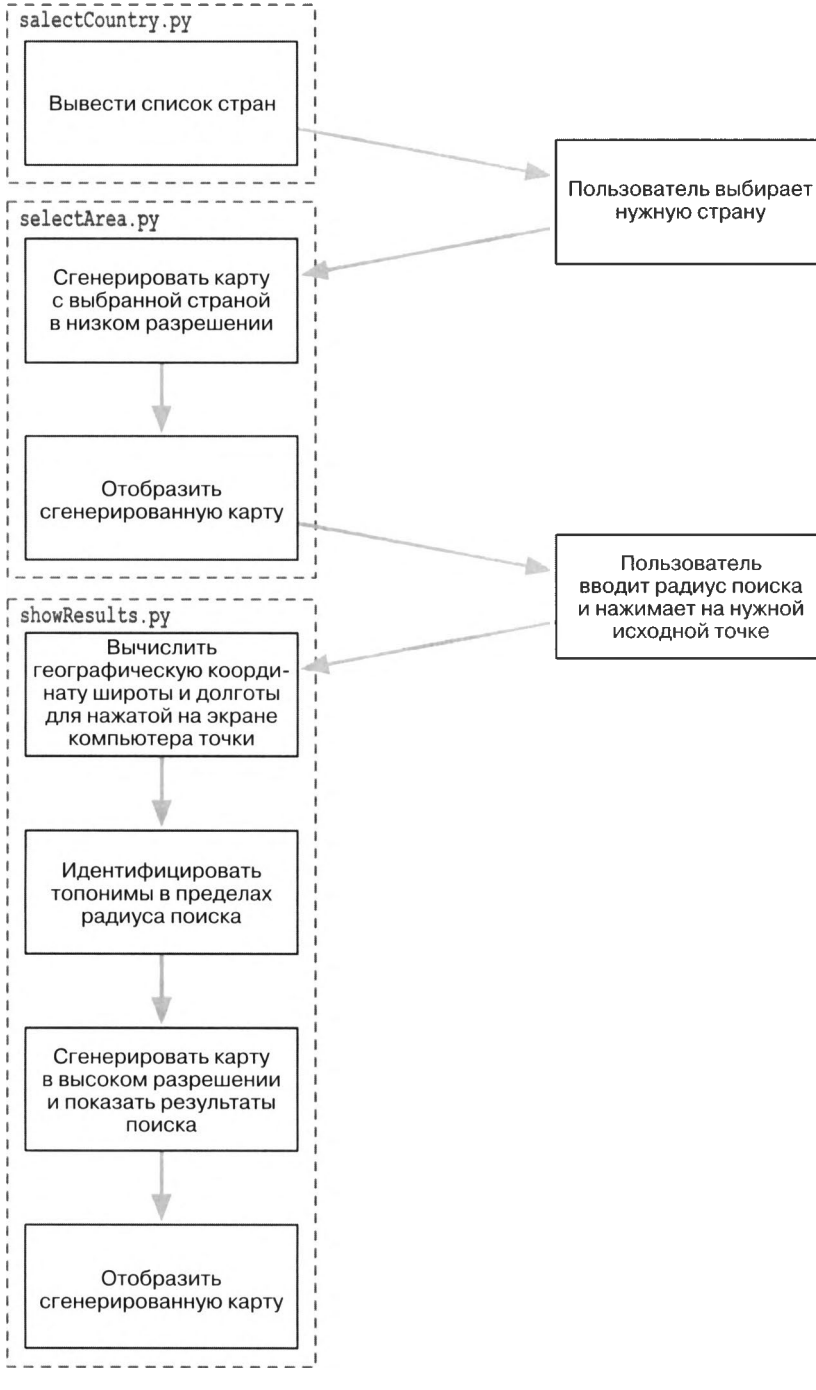
```
chmod +x selectCountry.py
```

На компьютерах под Windows расширение файла (`.py`) автоматически заставит сценарии CGI вызывать интерпретатор Python, поэтому вам не потребуется делать ничего из перечисленного.

Для получения дополнительной информации о сценариях CGI обратитесь к одному из учебных руководств по CGI, доступных в Интернете, например http://wiki.python.org/moin/CgiScripts.

Сценарии CGI – это не единственный способ, которым можно реализовать приложение DISTAL. Другие возможные подходы включают в себя использование платформ для разработки веб-приложений, таких как TurboGears или Django, использование AJAX для написания своего собственного динамического веб-приложения, применение CherryPy (http://cherrypy.org) или даже таких инструментов, как Pyjamas (http://pyjs.org), для компиляции программного кода Python в JavaScript. Все эти подходы, тем не менее, сложнее, чем реализация на основе CGI, и чтобы сделать программный код максимально прямолинейным, в этой главе мы будем использовать исключительно сценарии CGI.

Посмотрим, как наши сценарии CGI реализуют последовательность операций приложения DISTAL:


selectCountry.py
Вывести список стран
Пользователь выбирает нужную страну
selectArea.py
Сгенерировать карту с выбранной страной в низком разрешении
Отобразить сгенерированную карту
Пользователь вводит радиус поиска и нажимает на нужной исходной точке
showResults.py
Вычислить географическую координату широты и долготы для нажатой на экране компьютера точки
Идентифицировать топонимы в пределах радиуса поиска
Сгенерировать карту в высоком разрешении и показать результаты поиска
Отобразить сгенерированную карту

Как видите, есть три отдельных сценария CGI, selectcountry.py, selectArea.py и showResults.py, при этом каждый реализует обособленную часть приложения DISTAL.

Начнем с создания простого веб-сервера, способного выполнять наши сценарии CGI. В Python это делается легко; просто создайте следующую ниже программу, которую мы назовем webServer.py:

```
import http.server

PORT_NUMBER = 8000
address = ('', PORT_NUMBER)

try:
    handler = http.server.CGIHTTPRequestHandler
    server  = http.server.HTTPServer(address, handler)
    print('HTTP-сервер запущен на порту', PORT_NUMBER)
    server.serve_forever()
except KeyboardInterrupt:
    print('Получен управляющий символ ^C. Веб-сервер закрывается')
    server.socket.close()
```

Поместите этот файл в свой основной каталог DISTAL. Затем в том же каталоге создайте новый подкаталог с именем cgi-bin. Этот подкаталог будет содержать различные сценарии CGI, которые вы создадите.

Программа webServer.py запускает веб-сервер по адресу http://127.0.0.1:8000, который будет выполнять любые сценарии CGI, размещенные в подкаталоге cgi-bin. Так, например, для получения доступа к сценарию selectCountry.py в своем веб-браузере нужно ввести следующий URL-адрес:

```
http://127.0.0.1:8000/cgi-bin/selectCountry.py
```

Сценарий «выбрать страну»

Задача сценария selectCountry.py состоит в том, чтобы выводить пользователю список стран. Он позволит пользователю выбрать нужную страну, которая затем передается сценарию selectArea.py для последующей обработки.

Вот как будет выглядеть результат работы сценария `selectCountry.py`:

Нас сейчас не слишком волнует визуальное оформление нашей программы. Можно было бы легко воспользоваться таблицей стилей CSS или даже библиотекой, такой как Bootstrap, чтобы сделать систему DISTAL внешне более профессиональной. В этой главе мы прежде всего сконцентрируемся на функциональности, а не на том, как сделать нашу программу привлекательной.

Этот сценарий CGI очень прост – мы всего-навсего распечатываем содержимое страницы HTML, позволяющей пользователю выбрать страну из списка названий стран:

```
#!/usr/bin/env python3

import psycopg2

connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()

cursor.execute("SELECT id,name FROM countries ORDER BY name")

print('Content-Type: text/html; charset=utf8')
print()
print()
print('<html lang="ru">')
print('  <head><title>Выбрать страну</title></head>')
```

```
print('  <body>')
print('    <form method="POST" action="selectArea.py">')
print('      <p>')
print('      Выбрать страну.')
print('      </p><p>')
print('      <select name="countryID" size="12">')

for id, name in list(cursor):
    print('<option value="'+str(id)+'">'+name+'</option>')

print('      </select>')
print('      <p />')
print('      <input type="submit" value="Готово" />')
print('    </form>')
print('  </body>')
print('</html>')
```

Удостоверьтесь, что ваш сценарий называется `selectCountry.py` и помещен в каталог `cgi-bin`. Если вы выполняете его в Mac OS X или Linux, то вам нужно поменять строку шебанг в начале сценария, в случае если приходится использовать другой интерпретатор Python. Необходимо также воспользоваться командой `chmod`, как было описано ранее, чтобы сделать сценарий исполнимым.

Понимание HTML-форм

Если вы не использовали HTML-формы прежде, то паниковать не стоит. Они довольно прямолинейны, и если нужно, то можно просто скопировать приведенный здесь программный код примеров. Чтобы узнать о HTML-формах больше, загляните в одно из многих учебных руководств, доступных онлайн. Хороший пример можно найти на странице http://www.pagetutor.com/form_tutor.

Теперь у вас должно получиться выполнить этот сценарий, запустив программу `webserver.py` и введя в веб-браузере следующий URL-адрес:

```
http://127.0.0.1:8000/cgi-bin/selectCountry.py
```

Если все выполнено правильно, то вы должны увидеть список стран и сможете выбрать одну из них. Если щелкнуть по кнопке **Готово**, то вы должны увидеть страницу с кодом ошибки 404, обозначающей, что сценарий `selectArea.py` пока не существует, – что абсолютно верно, поскольку мы его еще не реализовали.

Сценарий «выбрать область»

Следующий компонент приложения DISTAL – это сценарий `selectArea.py`. Он генерирует веб-страницу, которая выводит простую карту выбранной страны. Пользователь может ввести нужный радиус поиска и щелкнуть по карте, чтобы определить начальную точку для выполнения поиска приложением DISTAL:

Поскольку этот сценарий влечет за собой отображение карты, нам придется научиться получать доступ к библиотеке Mapnik из сценария CGI. В *главе 7 «Генерирование карт при помощи Python и библиотеки Mapnik»* мы уже познакомились с тем, как работает библиотека Mapnik; для приложения DISTAL мы напишем автономный модуль `mapGenerator.py`, который генерирует карту и хранит получившееся изображение во временном файле, с тем чтобы вывести его на экран.

Вот полный исходный код модуля `mapGenerator.py`. Несмотря на то что этот скрипт не является сценарием CGI, он должен быть помещен в ваш каталог `cgi-bin`, для того чтобы различные сценарии CGI могли к нему обращаться.

```
# mapGenerator.py

import os, os.path, sys, tempfile
import mapnik
```

```
def generateMap(tableName, minX, minY, maxX, maxY,
                mapWidth, mapHeight,
                hiliteExpr=None, points=None):

  extent = "{},{},{},{}".format(minX, minY, maxX, maxY)

  layer = mapnik.Layer("Layer")
  layer.datasource = mapnik.PostGIS(dbname="distal",
                                    table=tableName,
                                    user="distal_user",
                                    password="...",
                                    extent=extent,
                                    geometry_field="outline",
                                    srid=4326)

  map = mapnik.Map(mapWidth, mapHeight,
                   '+proj=longlat +datum=WGS84')
  map.background = mapnik.Color("#8080a0")

  style = mapnik.Style()

  rule = mapnik.Rule()

  if hiliteExpr != None:
      rule.filter = mapnik.Filter(hiliteExpr)

  rule.symbols.append(mapnik.PolygonSymbolizer(
      mapnik.Color("#408000")))
  rule.symbols.append(mapnik.LineSymbolizer(
      mapnik.Stroke(mapnik.Color("#000000"), 0.1)))

  style.rules.append(rule)

  rule = mapnik.Rule()
  rule.set_else(True)
  rule.symbols.append(mapnik.PolygonSymbolizer(
      mapnik.Color("#a0a0a0")))
  rule.symbols.append(mapnik.LineSymbolizer(
      mapnik.Stroke(mapnik.Color("#404040"), 0.1)))

  style.rules.append(rule)

  map.append_style("Map Style", style)
  layer.styles.append("Map Style")
  map.layers.append(layer)

  if points != None:
      memoryDatasource = mapnik.MemoryDatasource()
      context = mapnik.Context()
      context.push("name")
      next_id = 1
      for long,lat,name in points:
          wkt = "POINT (%0.8f %0.8f)" % (long,lat)
          feature = mapnik.Feature(context, next_id)
```

```
            feature['name'] = name
            feature.add_geometries_from_wkt(wkt)
            next_id = next_id + 1
            memoryDatasource.add_feature(feature)

        layer = mapnik.Layer("Points")
        layer.datasource = memoryDatasource

        style = mapnik.Style()
        rule = mapnik.Rule()

        pointImgFile = os.path.join(os.path.dirname(__file__),
                                    "point.png")

        shield = mapnik.ShieldSymbolizer(
                   mapnik.Expression('[name]'),
                   "DejaVu Sans Bold", 10,
                   mapnik.Color("#000000"),
                   mapnik.PathExpression(pointImgFile))
        shield.displacement = (0, 7)
        shield.unlock_image = True
        rule.symbols.append(shield)

        style.rules.append(rule)

        map.append_style("Point Style", style)
        layer.styles.append("Point Style")

        map.layers.append(layer)

    map.zoom_to_box(mapnik.Envelope(minX, minY, maxX, maxY))
    scriptDir = os.path.dirname(__file__)
    cacheDir = os.path.join(scriptDir, "..", "mapCache")
    if not os.path.exists(cacheDir):
        os.mkdir(cacheDir)
    fd,filename = tempfile.mkstemp(".png", dir=cacheDir)
    os.close(fd)

    mapnik.render_to_file(map, filename, "png")

    return "../mapCache/" + os.path.basename(filename)
```

Напомним, что, для того чтобы программа могла получить доступ к базе данных, нужно указать пароль пользователя `distal_user`.

Значительная часть этого программного кода должна быть достаточно понятной, учитывая те понятия, которые были затронуты в *главе 7 «Генерирование карт при помощи Python и библиотеки Mapnik»*. Отметим только три вещи, которые вам могли еще не встречаться ранее:

- использование символизатора знаков `ShieldSymbolizer` для отображения точки;

- передача дополнительных параметров источнику данных PostGIS, требующихся в связи с тем, что библиотека Mapnik не может обрабатывать столбцы geography автоматически;
- использование временного файла, в котором хранится сгенерированное изображение карты, с тем чтобы его можно было показать на веб-странице. Как видите, мы возвращаем путь сгенерированного изображения карты относительно нашего главного каталога DISTAL.

Ниже приведен исходный код модуля mapGenerator.py, который выполняется в среде Python 3 под Fedora 24 и Mac OS X, с использованием декларативной парадигмы на основе XML:

```
#!/usr/bin/env python3
import os, tempfile
import mapnik

ROOT = os.path.dirname(__file__)

def generateMap(tableName,
                minX, minY, maxX, maxY,
                mapWidth, mapHeight,
                hiliteExpr=None,
                points=None):

    extent = "{},{},{},{}".format(minX, minY, maxX, maxY)

    # Определение карты
    map_string = '''<?xml version="1.0" encoding="utf-8"?>
<Map background-color="#8080a0" srs="+proj=longlat +datum=WGS84">
    <FontSet name="bold-fonts">
        <Font face-name="DejaVu Sans Bold" />
    </FontSet>
    <Style name="Map Style">
        <Rule>
            <!--(Filter)-->
            <PolygonSymbolizer fill="#408000" />
            <LineSymbolizer stroke="#000000" stroke-width="0.1" />
        </Rule>
        <Rule>
            <ElseFilter />
            <PolygonSymbolizer fill="#a0a0a0" />
            <LineSymbolizer stroke="#404040" stroke-width="0.1" />
        </Rule>
    </Style>
    <Style name="Point Style">
        <Rule>
            <ShieldSymbolizer
                face-name="DejaVu Sans Bold"
                fill="#000000"
                size="10"
```

```
                placement="point"
                file="(ImagePath)"
                unlock-image="true"
                shield-dx="0" shield-dy="-8">
                [name]
            </ShieldSymbolizer>
        </Rule>
    </Style>
    <Layer name="Layer">
        <StyleName>Map Style</StyleName>
        <Datasource>
            <Parameter name="type">postgis</Parameter>
            <Parameter name="dbname">distal</Parameter>
            <Parameter name="table">(TableName)</Parameter>
            <Parameter name="user">distal_user</Parameter>
            <Parameter name="password">...</Parameter>
            <Parameter name="extent">(Extent)</Parameter>
            <Parameter name="geometry_field">outline</Parameter>
            <Parameter name="srid">4326</Parameter>
        </Datasource>
    </Layer>

    <!--(PointLayer)-->
</Map>
'''
    #  Выполнение подстановок
    if hiliteExpr != None:   # отфильтровать по ИД страны = "[id] = 99"
        map_string = \
           map_string.replace("<!--(Filter)-->", "<Filter>" +
                                                 hiliteExpr +
                                                 "</Filter>")
    map_string = map_string.replace("(ImagePath)",
                    os.path.join(ROOT, "..", "images", "point.png"))
    map_string = map_string.replace("(TableName)", tableName)
    map_string = map_string.replace("(Extent)", extent)

    if points != None:  # перечислить точки и записать их в csv
        next_id = 1     # с разделителем символом "|" в слой Points
        csv = "id|name|wkt"
        for long,lat,name in points:
            csv = csv + "\n" + str(next_id) +
                  "|" + name + "|POINT (%0.8f %0.8f)" % (long,lat)
            next_id = next_id + 1
        wkt_src = '''<Layer name="Points">
                        <StyleName>Point Style</StyleName>
                        <Datasource>
                            <Parameter name="type">csv</Parameter>
                            <Parameter name="inline">(CSV)</Parameter>
                        </Datasource>
```

```
                </Layer>'''
        wkt_src = wkt_src.replace("(CSV)", csv)
        map_string = \
            map_string.replace("<!--(PointLayer)-->", wkt_src)

    # Визуализация карты
    gmap = mapnik.Map(mapWidth, mapHeight)
    mapnik.load_map_from_string(gmap, map_string)

    # сохранить снимок данных в xml-файл
    # mapnik.save_map(gmap, "distal_output_datafile.xml")
    gmap.zoom_to_box(mapnik.Envelope(minX, minY, maxX, maxY))

    cacheDir = os.path.join(ROOT, "..", "mapCache")
    if not os.path.exists(cacheDir):
        os.mkdir(cacheDir)
    fd,filename = tempfile.mkstemp(".png", dir=cacheDir)
    os.close(fd)

    mapnik.render_to_file(gmap, filename, "png")
    return "../mapCache/" + os.path.basename(filename)
```

Чтобы воспользоваться этим модулем, необходимо скачать или создать небольшой графический файл, который будет изображать положение топонима на карте. Оно представляет собой значок размера 9×9 пикселов, который выглядит следующим образом:

•

Копия этого значка включена в состав исходного кода примера, который прилагается к этой книге. Если у вас нет доступа к программному коду примера, можно создать или отыскать изображение, похожее на это; удостоверьтесь, что изображение называется `point.png` и помещено в тот же самый каталог, что и модуль `mapGenerator.py`.

Теперь мы готовы приступить к написанию сценария `selectArea.py`. Мы начнем со строки шебанг и импортируем различные модули, в которых мы будем нуждаться:

```
#!/usr/bin/env python3

import cgi, os.path, sys
import psycopg2
import mapGenerator
```

Затем определим некоторые полезные константы, чтобы облегчить генерирование HTML-кода:

```
HEADER = "Content-Type: text/html; charset=windows-1251\n\n" \
       + "<html><head><title>Выбрать участок</title>" \
       + "</head><body>"
FOOTER = "</body></html>"
```

```
HIDDEN_FIELD = '<input type="hidden" name="{}" value="{}">'

MAX_WIDTH  = 600
MAX_HEIGHT = 400
```

Потом откроем соединение с базой данных:

```
connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()
```

Наша следующая задача состоит в том, чтобы извлечь идентификатор страны, на которой пользователь нажал кнопкой мыши:

```
form = cgi.FieldStorage()
if "countryID" not in form:
    print(HEADER)
    print('<b>Пожалуйста, выберите страну</b>')
    print(FOOTER)
    sys.exit(0)

countryID = int(form['countryID'].value)
```

Располагая идентификатором выбранной страны, мы готовы начать генерирование карты. Этот процесс подразделяется на три этапа:

1. Вычисление ограничительной рамки, которая задает участок планеты, который будет выведен на экран компьютера.
2. Вычисление размеров карты.
3. Визуализация карты.

Рассмотрим каждый из них по очереди.

Вычисление ограничительной рамки

Прежде чем мы сможем показать выбранную страну на карте, мы должны вычислить для нее ограничительную рамку, то есть минимальное и максимальное значения широты и долготы. Данные об ограничительной рамке позволяют нам отобразить карту с нужной страной по центру.

Добавьте следующий ниже фрагмент в конец своего сценария `selectArea.py`:

```
cursor.execute("SELECT name," +
               "ST_YMin(ST_Envelope(outline)),"  +
               "ST_XMin(ST_Envelope(outline)),"  +
               "ST_YMax(ST_Envelope(outline)),"  +
               "ST_XMax(ST_Envelope(outline)) "  +
               "FROM countries WHERE id=%s",
               (countryID,))
row = cursor.fetchone()

if row != None:
    name    = row[0]
```

```
    min_lat = row[1]
    min_long = row[2]
    max_lat  = row[3]
    max_long = row[4]
else:
    print(HEADER)
    print('<b>Страна отсутствует</b>')
    print(FOOTER)
    sys.exit(0)

minLong = minLong - 0.2
minLat  = minLat - 0.2
maxLong = maxLong + 0.2
maxLat  = maxLat + 0.2
```

Как видите, мы извлекаем ограничительную рамку страны из базы данных и затем немного увеличиваем границы так, чтобы контуры страны не упиралась в края карты. Одновременно с этим мы также извлекаем название страны, поскольку оно нам вскоре понадобится.

Вычисление размеров карты

Поскольку страны имеют не квадратную форму, мы не можем просто нанести страну на карту фиксированного размера. Вместо этого мы должны вычислить **соотношение сторон** ограничительной рамки страны (отношение ширины рамки к ее высоте) и затем на основе этого соотношения сторон вычислить размер изображения географической карты, при этом ограничивая полный размер изображения так, чтобы она могла уместиться в пределах нашей веб-страницы. Вот необходимый код, который следует добавить в конец сценария `selectArea.py`:

```
width  = float(maxLong - minLong)
height = float(maxLat - minLat)
aspectRatio = width/height

mapWidth  = MAX_WIDTH
mapHeight = int(mapWidth / aspectRatio)

if mapHeight > MAX_HEIGHT:
    # Масштабировать карту до нужных размеров.
    scaleFactor = float(MAX_HEIGHT) / float(mapHeight)
    mapWidth = int(mapWidth * scaleFactor)
    mapHeight = int(mapHeight * scaleFactor)
```

Выполнение этого фрагмента будет означать, что карта будет оптимально подогнана по размеру в соответствии с размерами страны, которую мы выводим на экран компьютера.

Визуализация карты

При наличии заданной ограничительной рамки, размеров карты и источника данных мы, наконец, готовы визуализировать карту и сохранить ее в графическом

файле. Это выполняется следующим ниже образом при помощи вызова единственной функции:

```
hilite = "[id] = " + str(countryID)
imgFile = mapGenerator.generateMap("countries",
                                   minLong, minLat,
                                   maxLong, maxLat,
                                   mapWidth, mapHeight,
                                   hiliteExpr=hilite)
```

Отметим, что мы назначаем параметру hiliteExpr значение [id] = "+str (countryiD). Этот оператор визуально выделяет страну с заданным идентификатором.

Функция mapGenerator.generateMap() возвращает ссылку на графический файл в формате PNG, содержащий сгенерированную карту. Этот графический файл хранится во временном каталоге, и относительный путь к этому файлу возвращается вызывающей стороне. Это дает возможность использовать возвращенный графический файл imgFile непосредственно внутри нашего сценария CGI, как показано ниже:

```
print(HEADER)
print('<b>' + name + '</b>')
print('<form method="POST" action="showResults.py">')
print('  <p>')
print('  Выбрать все геообъекты в пределах')
print('  <input type="text" name="radius" value="10" size="2">')
print('  миль от точки.')
print('  </p><p>')
print('  Нажмите на карте, чтобы идентифицировать исходную точку:')
print('  <br>')
print('  <input type="image" src="' + imgFile + '" ismap>')
print(HIDDEN_FIELD.format("countryID", countryID))
print(HIDDEN_FIELD.format("countryName", name))
print(HIDDEN_FIELD.format("mapWidth", mapWidth))
print(HIDDEN_FIELD.format("mapHeight", mapHeight))
print(HIDDEN_FIELD.format("minLong", minLong))
print(HIDDEN_FIELD.format("minLat", minLat))
print(HIDDEN_FIELD.format("maxLong", maxLong))
print(HIDDEN_FIELD.format("maxLat", maxLat))
print('</p></form>')
print(FOOTER)
```

Шаблон HIDDEN_FIELD генерирует строки, которые выглядят следующим образом:

```
<input type="hidden" name="..." value="...">
```

Эти строки задают *скрытые поля формы*, которые передают различные значения, обрабатываемые в следующем сценарии CGI. То, каким образом эта информация используется, мы обсудим в следующем разделе.

Применение в этом сценарии CGI строки <input type="image" src="..." ismap> приводит к интересному результату – активации карты **нажатием кнопкой мыши**: когда пользователь щелкает по изображению, вложенная HTML-форма отправляется на сервер с двумя дополнительными параметрами x и y. Они содержат прямоугольную координату внутри изображения, по которому пользователь нажал кнопкой мыши.

На этом сценарий CGI selectArea.py завершен. Если вы выполняете его в Mac OS X либо Linux-машине, то вам придется удостовериться, что в начало программы вы добавили соответствующую строку шебанг и сделали программу исполнимой, как было описано ранее, с тем чтобы она могла выполняться в качестве сценария CGI.

Если все выполнено правильно, то вы сможете направить веб-браузер по адресу http://127.0.0.1:8000/cgi-bin/selectCountry.py.

При выборе страны вы должны увидеть карту этой страны, которая будет показана внутри веб-браузера. Если вы щелкнете внутри карты, то получите код ошибки 404, который обозначает, что заключительный сценарий CGI еще не написан.

Сценарий «показать результаты»

Заключительный сценарий CGI – это то место, где выполняется реальная работа. Начнем с создания файла showResults.py и наберем туда следующий ниже программный код:

```
#!/usr/bin/env python3

import psycopg2
import cgi
import mapGenerator

########################################################

MAX_WIDTH = 800
MAX_HEIGHT = 600
METERS_PER_MILE = 1609.344

########################################################

connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()
```

Напомним, что в оператор psycopg2.connect() следует ввести пароль для пользователя базы данных distal_user. Кроме того, следует пометить этот файл как исполнимый и ввести соответствующее значение шебанг в случае, если он выполняется в Mac OS X или Linux.

В этом сценарии мы возьмем прямоугольную координату (x, y) точки в месте нажатия пользователем кнопкой мыши и введенный радиус поиска, конвертируем прямоугольную координату (x, y) в долготу и широту и определим все топонимы в пределах заданного радиуса поиска. Затем мы сгенерируем карту с высоким раз-

решением, отобразив береговые линии и топонимы в пределах радиуса поиска, и покажем эту карту пользователю.

Напомним, что x соответствует значению долготы, а y – значению широты. Значение (x, y) эквивалентно значению (долгота, широта), а не (широта, долгота).

Проанализируем каждый из этих этапов по очереди.

Определение точки в месте нажатия

Сценарий selectArea.py генерирует HTML-форму, которая отправляется на сервер, когда пользователь нажимает кнопкой мыши на карте страны с низким разрешением. Сценарий showResults.py получает параметры формы, включая прямоугольные координаты x и y точки, по которой пользователь нажал.

Взятая отдельно, эта координата не имеет особой практической пользы, поскольку она просто-напросто представляет собой смещение точки в пикселах по осям X и Y в месте нажатия пользователем кнопкой мыши. Мы должны перевести предоставленную прямоугольную координату (x, y) пиксела в значение широты и долготы на поверхности Земли, которое соответствует точке в месте нажатия.

Для этого у нас должна быть следующая информация:

- ограничительная рамка карты в географических координатах: minLong, minLat, maxLong и maxLat;
- размер карты в пикселах: mapWidth и mapHeight.

Все эти переменные были рассчитаны в предыдущем разделе и переданы нам при помощи скрытых переменных формы вместе с названием и идентификатором страны, нужным радиусом поиска и прямоугольной координатой (x, y) точки в месте нажатия. Мы можем извлечь их при помощи модуля cgi. С этой целью добавьте в конец вашего файла showResults.py следующий ниже фрагмент:

```
form = cgi.FieldStorage()

countryID   = int(form['countryID'].value)
radius      = int(form['radius'].value)
x           = int(form['x'].value)
y           = int(form['y'].value)
mapWidth    = int(form['mapWidth'].value)
mapHeight   = int(form['mapHeight'].value)
countryName = form['countryName'].value
minLong     = float(form['minLong'].value)
minLat      = float(form['minLat'].value)
maxLong     = float(form['maxLong'].value)
maxLat      = float(form['maxLat'].value)
```

Имея в распоряжении эту информацию, теперь мы можем вычислить широту и долготу того места, где пользователь нажал кнопкой мыши. Начнем с вычисления того, как далеко по горизонтали расположено место нажатия пользователем на изображении. Это число будет в диапазоне между 0 и 1:

```
xFract = float(x)/float(mapWidth)
```

Значение xFract, равное 0.0, соответствует левой стороне изображения, тогда как значение 1.0 – правой стороне. Затем мы объединим его с минимальным и максимальным значениями долготы, чтобы вычислить долготу точки в месте нажатия:

```
longitude = minLong + xFract * (maxLong-minLong)
```

Потом мы проделаем то же самое, чтобы перевести координату *y* в значение широты:

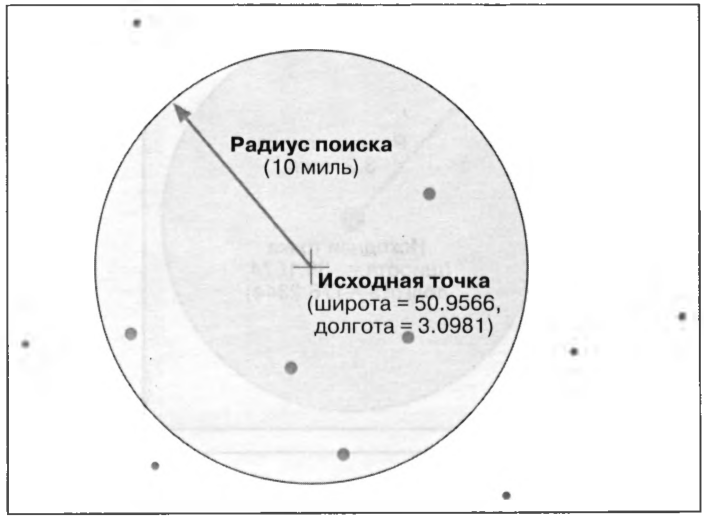
```
yFract = float(y)/float(mapHeight)
latitude = minLat + (1-yFract) * (maxLat-minLat)
```

Отметим, что в приведенной выше калькуляции вместо yFract мы используем (1-yFract). Это вызвано тем, что значение minLat относится к широте *основания* изображения, в то время как значение yFract, равное 0.0, соответствует его *вершине*. Применив выражение (1-yFract), мы переворачиваем значения вертикально, с тем чтобы рассчитать широту правильно.

Идентификация совпадающих топонимов

Подытожим достигнутое на данный момент: пользователь выбрал страну, просмотрел простую карту контура страны, ввел нужный радиус поиска и нажал на карте, чтобы определить начальную точку для поиска. Затем мы конвертировали точку в месте нажатия в значение долготы и широты.

В итоге мы получим три числа: нужный радиус поиска, а также широту и долготу точки, от которой можно начинать поиск. Теперь наша задача состоит в том, чтобы определить топонимы в пределах заданного радиуса поиска вокруг точки в месте нажатия:



К счастью, ввиду того, что в нашей таблице places есть столбец с типом GEOGRAPHY, мы можем поручить PostGIS сделать всю тяжелую работу за нас. Вот необходимый программный код для поиска соответствующих топонимов:

```
cursor.execute("SELECT ST_X(position::geometry)," +
               "ST_Y(position::geometry),name " +
               "FROM places WHERE " +
               "ST_DWithin(position, ST_MakePoint(%s, %s), %s)",
               (longitude, latitude, radius * METERS_PER_MILE))
points = []
for name,long,lat in cursor:
    points.append([long, lat, name])
```

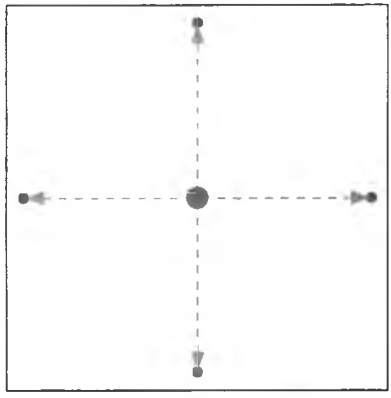
Отметим, что для извлечения значений X и Y из топонимов мы должны использовать синтаксическую конструкцию `position::geometry`. **Этот синтаксис переводит значение местоположения** `position` **из значения с типом** `geography` **назад в** `geometry` **– мы должны это сделать, потому что функции** `ST_X()` **и** `ST_Y()` **не принимают значения с типом** `geography`.

Соответствующие топонимы будут сохранены в переменной `points` как список кортежей (`долгота, широта, название`).

Отображение результатов

Вычислив список топонимов в пределах нужного радиуса поиска, теперь мы можем воспользоваться модулем `mapGenerator.py` для их отображения. Для этого нам нужно вычислить ограничительную рамку отображаемой области на основе точки в месте нажатия, предоставленном радиусе поиска и *обрамления*, которое обеспечит, что топонимы будут наноситься не на самом краю карты:

Для этого мы воспользуемся библиотекой `pyproj`, которая рассчитает положение четырех координат строго к северу, югу, востоку и западу от исходной точки:



То, насколько далеко мы продвинемся к северу, югу, востоку или западу, зависит от радиуса поиска, причем к этому расстоянию мы также добавим 10%-ный припуск. При помощи библиотеки pyproj эти расчеты довольно прямолинейные:

```
geod = pyproj.Geod(ellps="WGS84")
distance = radius * METERS_PER_MILE * 1.1

x,y,angle = geod.fwd(longitude, latitude, 0, distance)
maxLat = y

x,y,angle = geod.fwd(longitude, latitude, 90, distance)
maxLong = x

x,y,angle = geod.fwd(longitude, latitude, 180, distance)
minLat = y

x,y,angle = geod.fwd(longitude, latitude, 270, distance)
minLong = x
```

После того как мы это сделаем, мы можем воспользоваться нашим модулем mapGenerator.py, чтобы сгенерировать карту и показать найденные результаты:

```
imgFile = mapGenerator.generateMap("shorelines",
                                   minLong, minLat,
                                   maxLong, maxLat,
                                   600, 600,
                                   points=points)
```

Когда мы до этого вызывали генератор карты, для выделения конкретных геообъектов мы использовали выражение фильтрации. В данном случае нам ничего выделять не нужно. Вместо этого в именованном параметре points мы передаем генератору список топонимов для нанесения на карту.

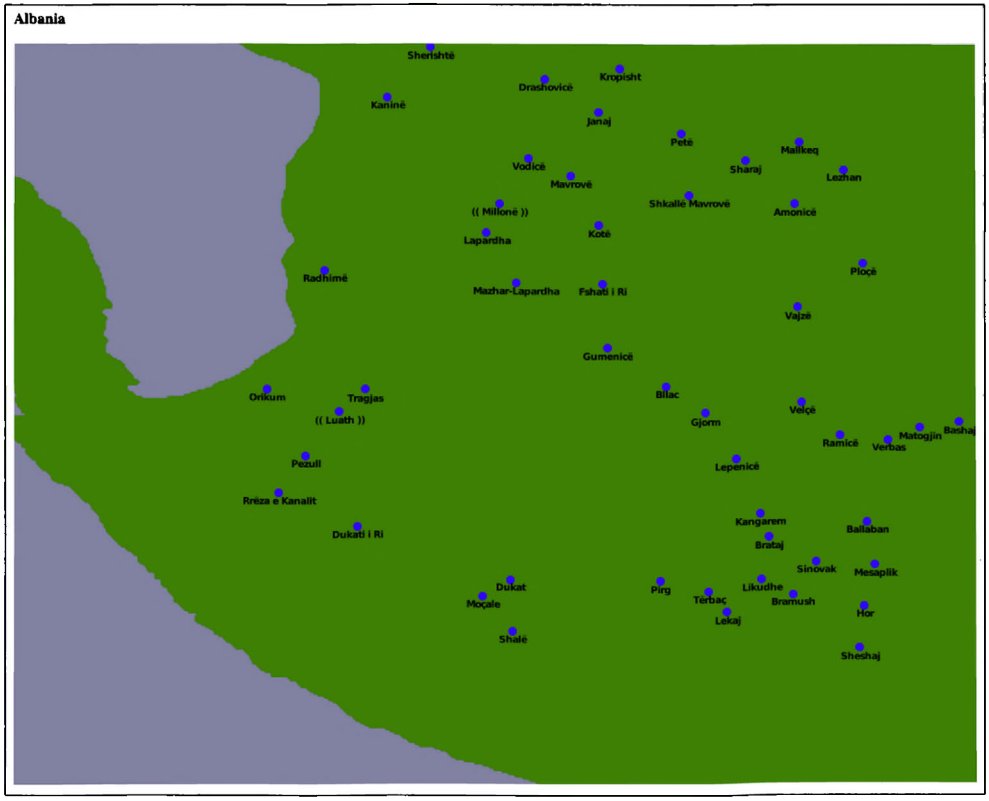
Генератор карты создает файл в формате PNG и возвращает ссылку на тот файл. Закончим нашу программу выводом карты на экран компьютера, с тем чтобы пользователь смог ее увидеть:

```
print(HEADER)
print('<b>' + countryName + '</b>')
print('<p>')
print('<img src="' + imgFile + '">')
print(FOOTER)
```

На этом наша первая попытка создания приложения DISTAL в целом завершена. Теперь посмотрим, как оно работает.

Использование приложения DISTAL

Чтобы выполнить систему DISTAL, просто запустите написанный вами ранее сценарий сервера `webserver.py` и воспользуйтесь своим веб-браузером, чтобы перейти по следующему URL-адресу: `http://127.0.0.1:8000/cgi-bin/selectCountry.py`. Выберите интересующую вас страну и затем щелкните по кнопке **Готово**. Затем введите радиус поиска и щелкните по точке внутри карты этой страны. Если все выполнено правильно, то вы должны увидеть подробную карту участка, по которому вы нажали кнопкой мыши, где будут показаны различные названия населенных пунктов. Следующее ниже изображение показывает, как должен выглядеть результат:



Наша программа могла бы делать гораздо больше, а если копнуть поглубже, то вы начнете замечать некоторые связанные с ней проблемы. Не переживайте; в следующей главе мы изучим эти проблемы повнимательнее.

Заключение

В этой главе мы реализовали технологичное веб-приложение, которое выводит на экран береговые линии, города и озера в пределах заданного радиуса от исходной точки.

Это приложение стало стимулом для исследования многих важных концепций в рамках разработки геопространственных приложений, в том числе хранения больших объемов пространственной информации в базе данных, выполнения запросов к пространственным данным на основе расстояний и использования библиотеки Mapnik для визуализации карты на веб-странице. В следующей главе мы изучим приложение DISTAL более подробно и рассмотрим способы, которые позволят улучшить удобство его использования и его производительность.

Глава 9

Совершенствование приложения DISTAL

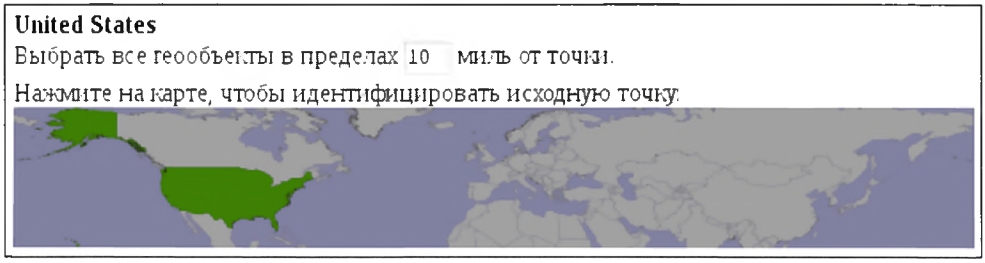
В предыдущей главе мы написали полнофункциональную реализацию системы DISTAL, которая работает, как и планировалось: пользователь может выбрать страну, ввести радиус поиска в милях, нажать на исходной точке и увидеть карту с высоким разрешением, показывающую все названия населенных пунктов в пределах нужного радиуса поиска. К сожалению, приложению DISTAL не хватает ряда аспектов, и даже поверхностный взгляд обнаруживает у него несколько основных проблем. В этой главе мы займемся их решением. По ходу мы:

- узнаем об антимеридиане и преодолеем трудности, которые он может вызывать при работе с географическими координатами на основе широты и долготы;
- рассмотрим, как добавить в наши сценарии CGI функционал масштабирования;
- узнаем о последствиях работы с крупными многоугольниками и о том, каким образом можно улучшить с ними работу, разделив их на более мелкие элементы;
- сделаем значительные усовершенствования в быстродействии нашего приложения DISTAL, разделив данные береговых линий на четырехугольные сегменты и выводя на экран по одному сегменту вместо всей карты береговых линий.

Разберем по очереди каждую из этих проблем.

Обработка линии антимеридиана

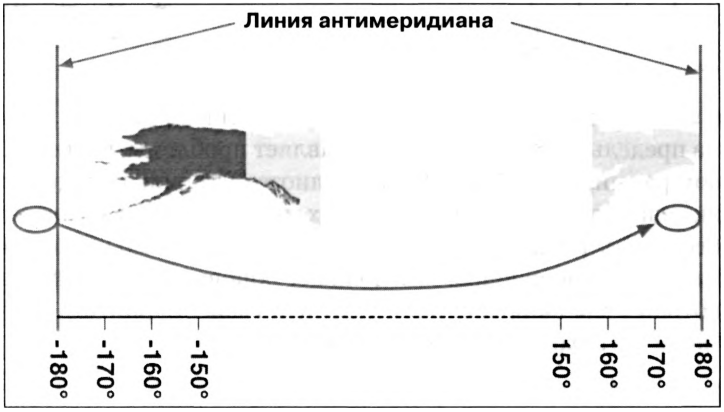
Если вы проанализируете приложение DISTAL, то вскоре обнаружите главную проблему, связанную с удобством его использования при работе с некоторыми странами. Например, если вы щелкнете по **United States** на странице **Выбрать страну**, то вам будет предложена следующая ниже карта, на которой нужно нажать, чтобы выбрать исходную точку:



Используя эту карту, точно нажать на нужной точке, наверное, не получится, потому что она охватывает бóльшую часть поверхности Земли.

Проблема заключается в том, что Аляска пересекает **линию антимеридиана**. Линия антимеридиана – это линия, где стыкуются левая и правая стороны карты мира, то есть в ±180° долготы. Учитывая то, каким образом значения долготы меняют знак и направление, двигаясь вокруг земного шара, значение –180° долготы эквивалентно значению +180° долготы. Пресловутая линия антимеридиана в ±180° долготы как раз и является причиной многих проблем при работе с геоданными.

В случае США часть полуострова Аляска простирается за пределы 180° западной долготы и уходит дальше через Алеутские острова, заканчиваясь на острове Атту, чья долгота равна 172° восточной долготы. Так как п/о Аляска пересекает линию антимеридиана, он появляется как на левой, так и на правой сторонах карты мира:

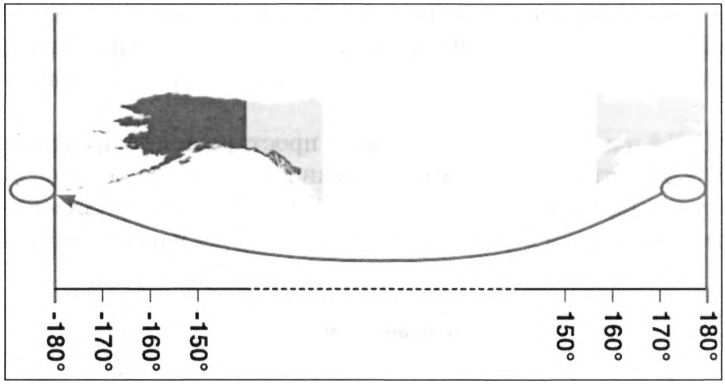


Поскольку участок п/о Аляска, который лежит левее от линии в –180°, смещается в правую сторону карты, приложение DISTAL ошибочно считает, что п/о Аляска и, следовательно, все США охватывают почти весь земной шар. Именно по этой причине мы получаем странные результаты, когда отображаем карту, охватывающую границы США.

Чтобы решить эту проблему, нам придется идентифицировать страны, которые пересекают линию антимеридиана, и корректировать их таким образом, чтобы они появлялись только на одной стороне линии.

В приложении DISTAL это можно сделать, потому что мы показываем контур всего одной страны. Если бы мы показывали сразу несколько стран, то нам пришлось бы создать две версии контуров каждой страны: одну в ее исходном месте, а другую смещенной на 180°. И затем мы могли бы выбирать, какой из двух наборов контуров отображать, исходя из выбранной пользователем страны.

Эта корректировка выполняется путем перемещения отдельных многоугольников, которые составляют контур страны таким образом, чтобы они все лежали по одну сторону карты. Получившиеся многоугольники будут лежать на одной из сторон от линии антимеридиана, одни – на западной стороне, а другие – на восточной:



Отметим, что эта корректировка перемещает полные границы многоугольников страны за пределы ±180°. Это не представляет проблемы, поскольку в геопространственном расширении PostGIS и в библиотеке Mapnik координаты рассматриваются как вещественные числа, у которых может быть любое значение.

Чтобы обеспечить поддержку корректировки многоугольников, которые пересекают линию антимеридиана, мы должны обновить сценарий `import_world_borders.py`, который мы написали в предыдущей главе. Вернитесь к этому сценарию и добавьте в этот файл следующую ниже выделенную строку:

```
for i in range(layer.GetFeatureCount()):
    feature = layer.GetFeature(i)
    name = feature.GetField("NAME")
    wkt = feature.GetGeometryRef().ExportToWkt()
    wkt = adjust_for_antimeridian(name, wkt)

    cursor.execute("INSERT INTO countries (name,outline) " +
                   "VALUES (%s, ST_GeometryFromText(%s, 4326))",
                   (name, wkt))
```

```
    num_done = num_done + 1

connection.commit()

print("Импортировано {} стран".format(num_done))
```

Как видите, мы вызываем новую функцию коррекции по антимеридиану `adjust_for_antimeridian()`, которая выполняет всю тяжелую работу. Прежде чем мы приступим к ее написанию, добавьте в верхнюю часть своей программы следующие ниже операторы импорта:

```
import shapely.wkt
from shapely.geometry import MultiPolygon
from shapely.affinity import translate
```

Наша функция будет использовать эти импортированные модули. Затем нам нужно определить непосредственно саму функцию. Сразу после операторов импорта добавьте к своей программе следующую ниже строку:

```
def adjust_for_antimeridian(name, wkt):
```

Как видите, мы передаем в функцию название страны и ее контур в формате WKT. Название страны требуется для того, чтобы можно было сообщить пользователю, контур какой страны мы скорректировали. По завершении работы наша функция вернет исходный WKT-контур без изменений либо новый WKT-контур, содержащий скорректированный контур страны.

Начнем с того, что из предоставленного текста в формате WKT создадим объект `Geometry` библиотеки Shapely и затем проверим, не является ли он составным многоугольником MultiPolygon. Если контур составным многоугольником не является, то у страны отсутствует несколько частей, и поэтому никакой корректировки выполнять не нужно. Вот соответствующий программный код, который следует ввести сразу после строки с определением функции:

```
outline = shapely.wkt.loads(wkt)
if outline.geom_type != "MultiPolygon":
    print("Импортирую {}".format(name))
    return wkt
```

Отметим, что мы распечатываем название страны, чтобы пользователь знал, какую страну мы импортируем. И затем возвращаем исходный контур без изменений.

Далее нам нужно проигнорировать все страны, которые не лежат близко к линии антимеридиана с одной из сторон. Если страна не лежит близко к линии антимеридиана с одной из сторон, то она не перекроет линию, и мы можем эту страну проигнорировать. Вот соответствующий фрагмент программы:

```
minLong,minLat,maxLong,maxLat = outline.bounds
if minLong >= -160 or maxLong <= 160:
    print("Импортирую {}".format(name))
    return wkt
```

Наша следующая задача состоит в том, чтобы разделить контур страны на отдельные части, где каждая часть – это один многоугольник внутри контура составного многоугольника. Каждую часть мы помечаем, находится многоугольник ближе к линии антимеридиана по левую сторону или по правую сторону. Добавьте в конец своей функции adjust_for_antimeridian() следующий ниже фрагмент:

```
parts = []

for geom in outline.geoms:
    left = geom.bounds[0]
    right = geom.bounds[2]
    if left == -180 and right == +180:
        print("{} простирается на весь мир, поэтому не подлежит смещению.".format(name))
        return wkt

    distance_to_left_side = -(-180 - left)
    distance_to_right_side = 180 - right

    if distance_to_left_side < distance_to_right_side:
        side = "left"
    else:
        side = "right"

    parts.append({'side' : side,
                  'geom' : geom})
```

Отметим, что мы проверяем, простирается ли единственный многоугольник внутри контура на весь мир. Если это так, то мы возвращаем исходный контур без изменений, поскольку у нас нет возможности его скорректировать. Отметим также, что эта проверка относится только для Антарктиды.

Разделив контур страны на отдельные части, теперь воспользуемся этой информацией, чтобы решить, смещать всю страну в левую либо правую сторону карты. Для этого мы подсчитаем, сколько частей находится на каждой стороне карты, и сместим всю страну в ту сторону, где частей больше. Добавьте в конец своей функции следующий ниже фрагмент, который это выполняет:

```
num_on_left = 0
num_on_right = 0

for part in parts:
    if part['side'] == "left":
        num_on_left = num_on_left + 1
    else:
        num_on_right = num_on_right + 1

if num_on_left > num_on_right:
    print("Смещаем {} влево".format(name))
    shift_direction = "left"
else:
    print("Смещаем {} вправо".format(name))
    shift_direction = "right"
```

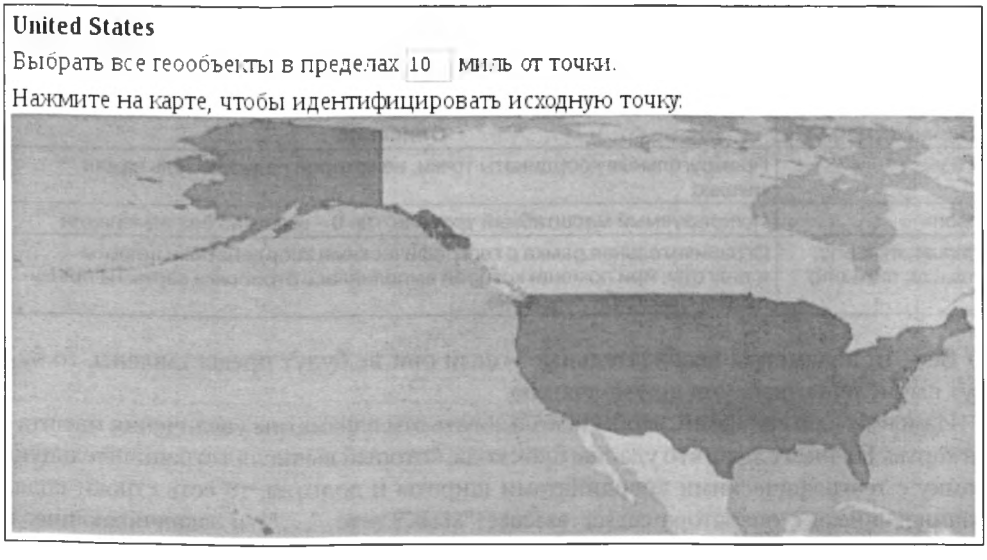
Затем нам нужно сместить части в соответствующую сторону. Иными словами, если мы смещаем всю страну вправо, то мы сместим вправо любые части, которые находятся слева, добавив 360° к долготе такой части. Точно так же, если мы смещаем страну влево, то мы сместим влево любые части, которые находятся справа, вычитая 360° из долготы такой части. Вот соответствующий фрагмент кода:

```
for part in parts:
    old_bounds = part['geom'].bounds
    if part['side'] == "left" and shift_direction == "right":
        part['geom'] = translate(part['geom'], 360)
    elif part['side'] == "right" and shift_direction == "left":
        part['geom'] = translate(part['geom'], -360)
```

И в заключение нам нужно снова соединить различные части в единый контур, который мы сможем вернуть вызывающей функции в формате WKT:

```
polygons = []

for part in parts:
    polygons.append(part['geom'])
combined = MultiPolygon(polygons)

return combined.wkt
```

На этом наша функция `adjust_for_antimeridian()` завершена. Теперь вы сможете повторно импортировать контуры стран, выполнив обновленный сценарий `import_world_borders.py`. Если вы затем запустите приложение DISTAL, то увидите, что контур США больше не охватывает весь земной шар:



Решение проблемы масштабирования

Приведенная выше иллюстрация обнаруживает вторую связанную с системой DISTAL проблему: поскольку США, включая Аляску, имеют протяженность более 4000 миль в ширину, попытки точно выбрать 10-мильный радиус поиска путем нажатия на точку, расположенную на этой карте, стали бы пустым занятием, доводящим до фрустрации, во всяком случае, разочарование вы бы испытали.

Чтобы решить эту проблему, мы реализуем функционал **масштабирования** (зуммирования), с тем чтобы пользователь мог точнее выбрать нужную исходную точку, нажав на ней мышью. Поскольку система DISTAL реализована как серия сценариев CGI, наш функционал масштабирования будет достаточно упрощенным: если пользователь, щелкая мышью, удерживает клавишу *Shift*, то мы увеличиваем масштаб точки в месте нажатия. Если во время щелчка мышью клавиша *Shift* не удерживается, то мы продолжаем поиск, как обычно.

В реальном веб-приложении мы бы реализовали полнофункциональный интерфейс скользящей карты, который поддерживает перетаскивание, а также экранные средства управления с целью увеличения и уменьшения масштаба. Выполнение этой работы находится за рамками того, что можно сделать при помощи простых сценариев CGI, как бы мы не хотели. Мы возвратимся к теме скользящих карт в *главе 10 «Инструменты для разработки геопространственных веб-приложений»*.

Чтобы реализовать масштабирование, мы должны обновить сценарий `selectArea.py`, который мы написали ранее. Для того чтобы обнаружить, удерживал пользователь клавишу *Shift* или нет, нам понадобится немного элементарного программного кода на JavaScript, и если клавиша нажата, то перезагрузим страницу «области выбора» с дополнительными параметрами CGI, которые позволяют увеличивать масштаб. В частности, мы изменим наш сценарий CGI, чтобы он мог принимать следующие ниже дополнительные параметры:

Параметр(ы) CGI	Описание
x и y	Прямоугольные координаты точки, на которой пользователь нажал мышью
zoom	Используемый масштабный уровень, где 0 – масштаб без изменений
minLat, minLong, maxLat, maxLong	Ограничительная рамка с географическими координатами широты и долготы, при помощи которой выполнялась отрисовка карты на предыдущем масштабном уровне

Все эти параметры необязательные – если они не будут предоставлены, то будут вычислены значения по умолчанию.

Изменим наш сценарий, чтобы использовать эти параметры увеличения масштаба карты. Начнем с того, что удалим блок кода, который вычислял ограничительную рамку с географическими координатами широты и долготы, то есть строки кода, начинающиеся с оператора `cursor.execute("SELECT name, ...")` и заканчивающиеся строкой `maxLat = maxLat + 0.2`. Поменяйте их на следующий ниже фрагмент кода:

```
if "x" in form:
    click_x = int(form['x'].value)
else:
    click_x = None

if "y" in form:
    click_y = int(form['y'].value)
else:
    click_y = None

if "zoom" in form:
    zoom = int(form['zoom'].value)
else:
    zoom = 0

if ("minLat" in form and "minLong" in form and
    "maxLat" in form and "maxLong" in form):
    # Использовать предоставленную ограничительную рамку.
    minLat  = float(form['minLat'].value)
    minLong = float(form['minLong'].value)
    maxLat  = float(form['maxLat'].value)
    maxLong = float(form['maxLong'].value)
else:
    # Рассчитать ограничительную рамку, исходя из контура страны.

    cursor.execute("SELECT " +
                   "ST_YMin(ST_Envelope(outline)),"  +
                   "ST_XMin(ST_Envelope(outline)),"  +
                   "ST_YMax(ST_Envelope(outline)),"  +
                   "ST_XMax(ST_Envelope(outline)) "  +
                   "FROM countries WHERE id=%s", (countryID,))

  row = cursor.fetchone()
  if row != None:
      minLat  = row[0]
      minLong = row[1]
      maxLat  = row[2]
      maxLong = row[3]
  else:
      print(HEADER)
      print('<b>Страна отсутствует</b>')
      print(FOOTER)
      sys.exit(0)

minLong = minLong - 0.2
minLat  = minLat - 0.2
maxLong = maxLong + 0.2
maxLat  = maxLat + 0.2

# Взять название страны.

cursor.execute("SELECT name FROM countries WHERE id=%s",
               (countryID,))
name = cursor.fetchone()[0]
```

Как видите, мы извлекаем параметры CGI x, y и zoom, и если они не были предоставлены, то присваиваем им значения по умолчанию. Затем мы проверяем наличие параметров minLat, minLong, maxLat и maxLong. В противном случае мы поручаем базе данных вычислить эти значения на основе контура страны и затем добавляем кромку 0.2 градуса по каждому. Наконец, мы загружаем название страны из базы данных.

Располагая параметрами CGI и при этом, когда нужно, вычисляя их значения по умолчанию, теперь мы можем использовать эти параметры для увеличения масштаба карты, как и требуется. К счастью, это довольно легко: если пользователь увеличил масштаб, то мы просто модифицируем ограничительную рамку так, чтобы показать меньший участок планеты. Для этого сразу перед строкой hilite = "[id] = " + str(countryID) добавьте следующий ниже фрагмент кода:

```
if zoom != 0 and click_x != None and click_y != None:
    xFract = float(click_x)/float(mapWidth)
    longitude = minLong + xFract * (maxLong-minLong)

    yFract = float(click_y)/float(mapHeight)
    latitude = minLat + (1-yFract) * (maxLat-minLat)

    width = (maxLong - minLong) / (2**zoom)
    height = (maxLat - minLat) / (2**zoom)

    minLong = longitude - width / 2
    maxLong = longitude + width / 2
    minLat = latitude - height / 2
    maxLat = latitude + height / 2
```

В этой части сценария мы сначала вычисляем широту и долготу точки, где пользователь нажал кнопкой мыши. Наша задача – увеличить масштаб этой точки. Затем мы используем масштабный уровень, чтобы вычислить ширину и высоту отображаемого участка, и потом пересчитываем ограничительную рамку так, чтобы она имела заданную ширину и высоту, но центрировалась на точке в месте нажатия.

Далее нам нужно написать код на JavaScript, который распознает, когда пользователь щелкнул мышью, удерживая клавишу *Shift*, и вызвать наш сценарий с соответствующими параметрами снова. Для этого замените определение переменной HEADER, расположенной ближе к началу файла, следующим ниже фрагментом:

```
HEADER = "\n".join([
  "Content-Type: text/html; charset=windows-1251",
  "",
  "",
  "<html><head><title>Выбрать участок</title>",
  "<script type='text/javascript'>",
  "  function onClick(e) {",
  "    e = e || window.event;",
  "    if (e.shiftKey) {",
  "      var target = e.target || e.srcElement;",
  "      var rect = target.getBoundingClientRect();",
  "      var offsetX = e.clientX - rect.left;",
```

```
  "      var offsetY = e.clientY - rect.top;",
  "      var countryID = document.getElementsByName('countryID')[0].
value;",
  "      var minLat = document.getElementsByName('minLat')[0].
value;",
  "      var minLong = document.getElementsByName('minLong')[0].
value;",
  "      var maxLat = document.getElementsByName('maxLat')[0].
value;",
  "      var maxLong = document.getElementsByName('maxLong')[0].
value;",
  "      var zoom = document.getElementsByName('zoom')[0].value;",
  "      var new_zoom = parseInt(zoom, 10) + 1;",
  "      window.location.href = 'selectArea.py'",
  "                           + '?countryID=' + countryID",
  "                           + '&minLat=' + minLat",
  "                           + '&minLong=' + minLong",
  "                           + '&maxLat=' + maxLat",
  "                           + '&maxLong=' + maxLong",
  "                           + '&zoom=' + new_zoom",
  "                           + '&x=' + offsetX",
  "                           + '&y=' + offsetY;",
  "      return false;",
  "    } else {",
  "      return true;",
  "    }",
  "  }",
  "</script>",
  "</head><body>"])
```

Мы не будем вдаваться в подробности этого фрагмента, так как синтаксис JavaScript выходит за рамки этой книги. В этом фрагменте мы определяем функцию под названием `onClick()`, которая проверяет, не удерживал ли пользователь клавишу *Shift*, и если это так, то она вычисляет различные параметры CGI и присваивает встроенной переменной `window.location.href` URL-адрес, при помощи которого сценарий CGI перезагружается с этими параметрами.

Чтобы завершить создание нашей новой опции масштабирования, осталось решить всего две задачи: нам нужно добавить новое скрытое поле формы, чтобы сохранить значение переменной `zoom`, и вызвать нашу функцию `onClick()`, когда пользователь нажимает на карте. Добавьте следующую ниже строку к перечню операторов `print(HIDDEN_FIELD.format(...))` ближе к концу сценария:

```
print(HIDDEN_FIELD.format("zoom", zoom))
```

И в заключение замените строку `print('<input type="image"...>')` на приведенную ниже:

```
print('<input type="image" src="' + imgFile + '" ismap ' +
      'onClick="return onClick()">')
```

На этом изменения, необходимые для поддержки опции масштабирования, завершены. Если теперь выполнить программу DISTAL и выбрать страну, то вы сможете увеличивать масштаб карты, одновременно удерживая клавишу *Shift* и нажав на карте мышью. Это должно намного облегчить выбор нужной точки внутри более крупной страны.

Чтобы снова уменьшить масштаб, щелкните в своем веб-браузере по кнопке **Back** (Назад).

Мы теперь исправили две главные проблемы, связанные с удобством использования системы DISTAL. Тем не менее есть еще один аспект, на который мы должны обратить внимание: производительность. Ею мы и займемся.

Производительность

Наше приложение DISTAL, конечно же, работает, но его производительность оставляет желать лучшего. В то время как сценарии `selectCountry.py` и `selectArea.py` выполняются быстро, выполнение сценария `showResults.py` может занять от двух до нескольких секунд. Ясно, что такой результат недостаточно хорош: такая задержка неприятна для пользователя, но имела бы катастрофические последствия для сервера, если бы он начал получать больше запросов, чем он может обработать.

Поиск проблемы

Добавив к сценарию `showResults.py` функционал хронометража, можно быстро найти узкое место:

```
Вычисление координаты широты и долготы заняло 0.0110 сек.
Идентификация топонимов заняла 0.0088 сек.
Генерирование карты заняло 3.0208 сек.
Конструирование HTML-страницы заняло 0.0000 сек.
```

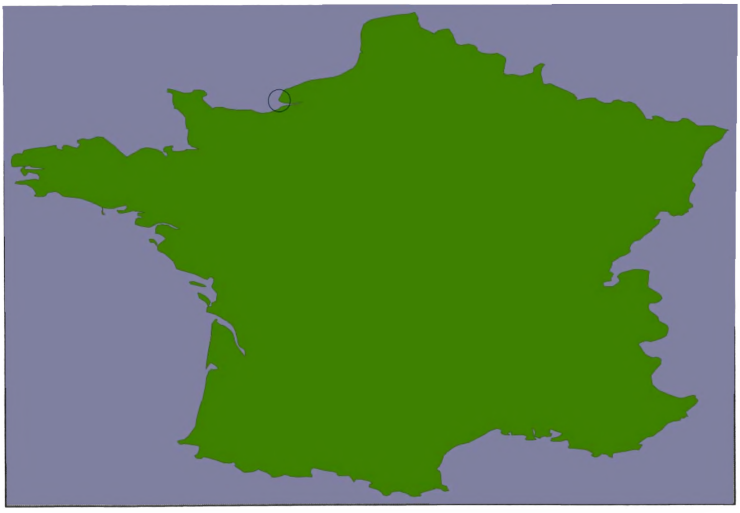
Ясно, что проблема здесь вызвана процессом генерирования карты. Поскольку генерирование карты в сценарии `selectArea.py` заняло доли секунды, то проблема такой медленной работы кроется совсем не в нем; нет ничего, характерного для процесса генерирования карты, что приводило бы к таким временным затратам. Тогда что же изменилось?

Вполне может быть, что много времени занимает отображение топонимов, но это маловероятно. Напротив, проблема с гораздо большей вероятностью может быть вызвана объемом данных географической карты, которые мы отображаем: сценарий `showResults.py` использует контуры береговых линий с высоким разрешением из набора географических данных GSHHG, в отличие от контуров стран с низким разрешением из набора данных границ стран мира. Чтобы проверить это предположение, можно поменять данные географической карты, при помощи которых генерируется карта, изменив спенарий `showResults.py` так, чтобы он использовал таблицу стран с низким разрешением вместо таблицы береговых линий с высоким разрешением.

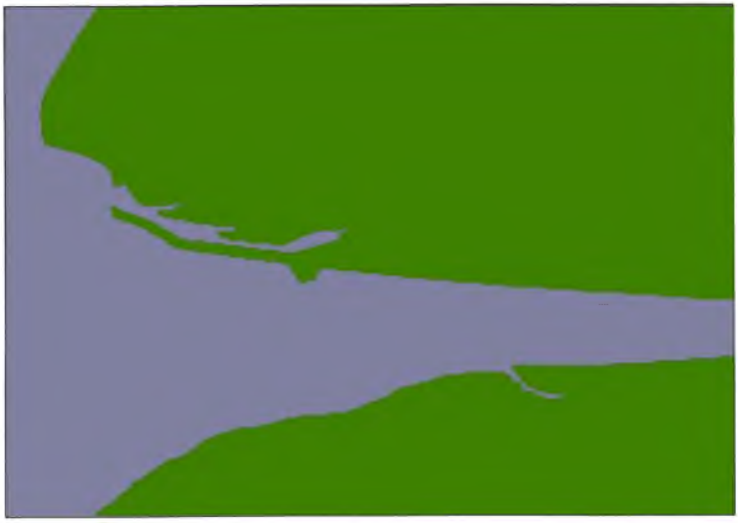
И в результате получим драматическое улучшение производительности:

```
Генерирование карты заняло 0.1729 сек.
```

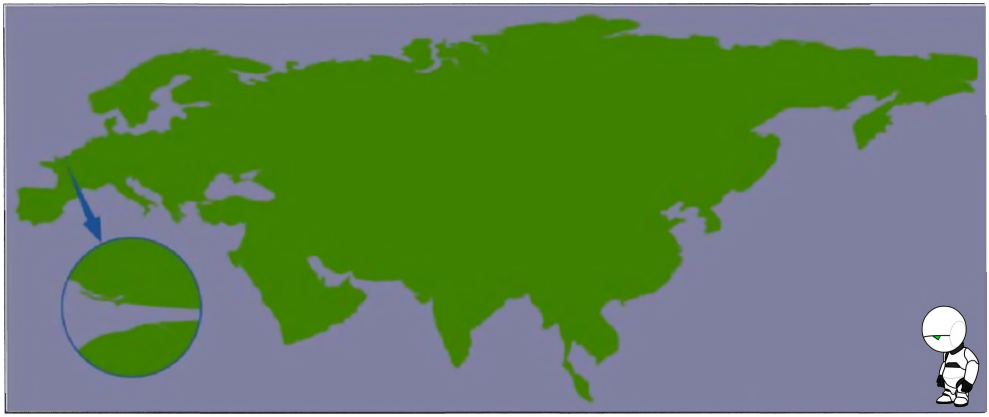
Тогда каким образом можно ускорить генерирование карты в сценарии `showResults.py`? Ответ заключается в природе данных береговых линий и в том, как мы их используем. Рассмотрим ситуацию, где вы идентифицируете точки в пределах 10 миль от г. Гавр во Франции:



Изображение береговой линии с высоким разрешением было бы аналогично следующему ниже:



Но этот участок береговой линии на самом деле является составной частью следующего объекта береговой линии географической базы данных GSHHG:



Этот многоугольник береговой линии огромен и состоит из более 1.1 миллиона точек, и мы отображаем всего лишь незначительную его часть.

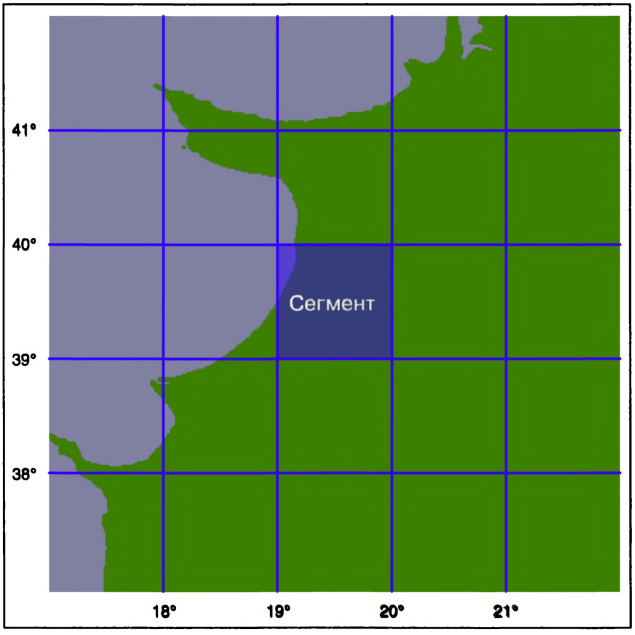
Поскольку эти многоугольники береговой линии такие большие, чтобы в итоге получить нужный участок береговой линии, генератору карты приходится считывать весь огромный многоугольник и затем отбрасывать из него 99% за ненадобностью. Кроме того, из-за очень крупного размера ограничительных рамок многоугольников при генерировании карты обрабатывается (и затем отфильтровывается) большое число лишних многоугольников. Именно поэтому сценарий `showResults.py` такой медленный.

С учетом трудностей, которые в работе с геоданными вызывают большие и сложные многоугольники, вам, скорее всего, понадобится рассмотренная здесь методика, которая поможет улучшить производительность ваших собственных систем.

Улучшение производительности

Разумеется, выполнение сценария `showResults.py` можно улучшить. Как было упомянуто в разделе наиболее успешных практических приемов *главы 6 «Пространственные базы данных»*, пространственные индексы работают лучше всего с относительно малыми геометриями – а наши многоугольники береговых линий совсем не маленькие. Однако, ввиду того,что приложение DISTAL показывает точки только в пределах определенного расстояния, мы можем нарезать эти огромные многоугольники на четырехугольные **сегменты** (tiles), которые еще именуются регулярными ячейками или клетками картографической сетки, предварительно их вычислить и сохранить в базе данных.

Допустим, мы ограничим радиус поиска 100 км. Кроме того, мы произвольно зададим сегмент так, чтобы он составлял один целый градус широты в высоту и один целый градус долготы в ширину:

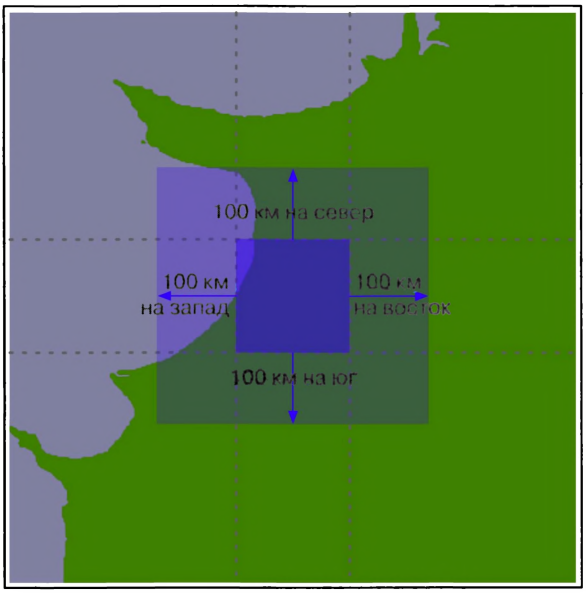


Отметим, что можно выбрать любой понравившийся размер сегмента, однако мы выбрали целые градусы долготы и широты, чтобы облегчить расчеты относительно того, внутри какого сегмента находится конкретная географическая координата широты и долготы. Каждому четырехугольному сегменту будет присвоено целочисленное значение широты и долготы, которые мы назовем `iLat` и `iLong`. Затем мы вычислим требующийся сегмент, исходя из любой заданной широты и долготы, как показано ниже:

```
iLat = int(round(latitude))
iLong = int(round(longitude))
```

Далее мы просто отыщем сегмент по заданному значению `iLat` и `iLong`.

Для каждого четырехугольного сегмента мы вычислим ограничительную рамку, а именно границы прямоугольника в 100 км к северу, востоку, западу и югу от сегмента:



Затем можно вычислить пересечение данных береговой линии с этой ограничительной рамкой:

В результате любой поиск, выполненный внутри границ четырехугольного сегмента, вплоть до максимальных 100 км в любом направлении, отобразит береговые

линии только в пределах этой ограничительной рамки. Мы сохраним пересеченную береговую линию в базе данных вместе с географическими координатами широты и долготы сегмента и поручим генератору цифровой карты для визуализации нужной береговой линии вместо исходных многоугольников географической базы данных GSHHG использовать контур соответствующего четырехугольного сегмента.

Вычисление сегментов береговых линий

Напишем программу, которая вычисляет четырехугольные сегменты береговых линий. Назовем ее `tileShorelines.py` и начнем с того, что добавим туда следующий фрагмент кода:

```
import math

import pyproj
from shapely.geometry import Polygon
from shapely.ops import cascaded_union
import shapely.wkt
import psycopg2

MAX_DISTANCE = 100000 # Максимальный радиус поиска в метрах.
```

Далее нам нужна функция, которая вычисляет ограничительные рамки сегментов. Функция `expandRect()` должна взять прямоугольник, заданный географическими координатами широты и долготы, и развернуть его в каждом направлении на заданное число метров. На основе освоенной нами методики это достаточно просто сделать: можно воспользоваться библиотекой `pyproj` для выполнения обратной калькуляции по дуге большого круга, чтобы вычислить четыре точки на заданном числе метров к северу, востоку, югу и западу от исходного прямоугольника. В результате мы получим нужную ограничительную рамку. Вот как будет выглядеть наша функция:

```
def expandRect(minLat, minLong, maxLat, maxLong, distance):
    geod = pyproj.Geod(ellps="WGS84")
    midLat = (minLat + maxLat) / 2.0
    midLong = (minLong + maxLong) / 2.0

    try:
        availDistance = geod.inv(midLong, maxLat, midLong,
                                 +90)[2]
        if availDistance >= distance:
            x,y,angle = geod.fwd(midLong, maxLat, 0, distance)
            maxLat = y
        else:
            maxLat = +90
    except:
        maxLat = +90 # расширение на север не получается.

    try:
        availDistance = geod.inv(maxLong, midLat, +180,
                                 midLat)[2]
```

```
        if availDistance >= distance:
            x,y,angle = geod.fwd(maxLong, midLat, 90, distance)
            maxLong = x
        else:
            maxLong = +180
    except:
        maxLong = +180 # расширение на восток не получается.

    try:
        availDistance = geod.inv(midLong, minLat, midLong,
                                 -90)[2]
        if availDistance >= distance:
            x,y,angle = geod.fwd(midLong, minLat, 180, distance)
            minLat = y
        else:
            minLat = -90
    except:
        minLat = -90 # расширение на юг не получается.

    try:
        availDistance = geod.inv(maxLong, midLat, -180,
                                 midLat)[2]
        if availDistance >= distance:
            x,y,angle = geod.fwd(minLong, midLat, 270, distance)
            minLong = x
        else:
            minLong = -180
    except:
        minLong = -180 # расширение на запад не получается.

    return (minLat, minLong, maxLat, maxLong)
```

Отметим, что мы добавили сюда проверку ошибок, чтобы учесть прямоугольники, лежащие близко к Северному или Южному полюсу.

При помощи этой функции мы сможем рассчитать ограничивающий прямоугольник для заданного сегмента следующим образом:

```
minLat,minLong,maxLat,maxLong = expandRect(iLat, iLong,
                                           iLat+1, iLong+1,
                                           MAX_DISTANCE)
```

Добавьте в ваш сценарий `tileShorelines.py` функцию `expandRect()`, поместив ее сразу за последним оператором импорта.

Теперь мы готовы приступить к реализации алгоритма нарезки на сегменты. Начнем работу, открыв соединение с базой данных:

```
connection = psycopg2.connect(database="distal",
                              user="distal_user",
                              password="...")
cursor = connection.cursor()
```

Напомним о необходимости указать пароль, который вы внесли для `distal_user`, когда настраивали базу данных.

Затем нам нужно загрузить в оперативную память многоугольники береговых линий. Вот необходимый программный код:

```
shorelines = []

cursor.execute("SELECT ST_AsText(outline) " +
               "FROM shorelines WHERE level=1")

for row in cursor:
    outline = shapely.wkt.loads(row[0])
    shorelines.append(outline)
```

Данная реализация алгоритма нарезки береговых линий на сегменты использует много оперативной памяти. Если ОЗУ вашего компьютера менее 2 Гб, то вам, скорее всего, придется сохранять временные результаты в базе данных. Выполнение этой операции, разумеется, замедлит процесс нарезки, но алгоритм все равно будет работать.

Загрузив многоугольники береговых линий, теперь можно приступить к вычислению содержимого каждого сегмента. Создадим список списков, который будет содержать (возможно, обрезанные) многоугольники внутри каждого сегмента; добавьте к концу вашего сценария `tileShorelines.py` следующий ниже фрагмент программы:

```
tilePolys = []
  for iLat in range(-90, +90):
      tilePolys.append([])
      for iLong in range(-180, +180):
          tilePolys[-1].append([])
```

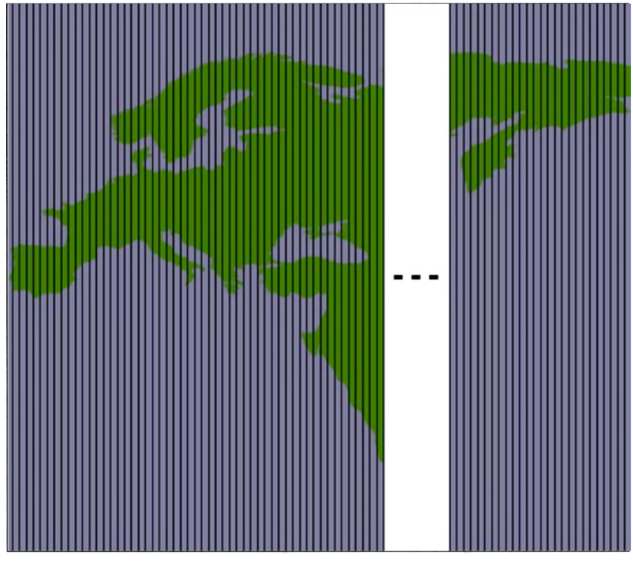
Для данной комбинации `iLat`/`iLong` сегмент `tilePolys[iLat][iLong]` будет содержать список многоугольников береговых линий, которые содержатся в этом сегменте.

Теперь нам нужно заполнить массив `tilePolys` участками береговых линий, которые появятся внутри каждого сегмента. Очевидное решение заключается в вычислении пересечений многоугольников, которое выполняется следующим образом:

```
shorelineInTile = shoreline.intersection(tileBounds)
```

К сожалению, этот подход к вычислениям занял бы *очень* продолжительное время – аналогично тому, как при генерировании карты приблизительно 2–3 секунды уходит на то, чтобы вычислить видимую часть береговых линий, потребуется приблизительно 2–3 секунды на то, чтобы выполнить это пересечение на огромном многоугольнике береговых линий. Имея 360×180 = 64 800 сегментов, выполнение расчетов на основе этого наивного подхода заняло бы несколько дней.

Намного более быстрое решение состоит в том, чтобы «разделять и властвовать» над большими многоугольниками. Мы сначала нарезаем огромный многоугольник береговых линий на вертикальные полосы, как показано ниже:



А затем нарезаем каждую вертикальную полосу горизонтально, чтобы получить отдельные участки многоугольника, которые можно объединять в отдельные сегменты:

В результате нарезки огромных многоугольников на полосы и затем дальнейшей нарезки каждой полосы процесс пересечения проходит намного быстрее. Вот фрагмент кода, который выполняет это пересечение; мы начинаем с итеративного просмотра каждого многоугольника береговых линий и вычисляем границы многоугольника, расширенные до ближайшего целого числа:

```
for shoreline in shorelines:
    minLong,minLat,maxLong,maxLat = shoreline.bounds
    minLong = int(math.floor(minLong))
    minLat  = int(math.floor(minLat))
    maxLong = int(math.ceil(maxLong))
    maxLat  = int(math.ceil(maxLat))
```

Потом нарезаем многоугольник на вертикальные полосы:

```
vStrips = []
for iLong in range(minLong, maxLong+1):

  stripMinLat  = minLat
  stripMaxLat  = maxLat
  stripMinLong = iLong
  stripMaxLong = iLong + 1

bMinLat,bMinLong,bMaxLat,bMaxLong = \
        expandRect(stripMinLat, stripMinLong,
                   stripMaxLat, stripMaxLong,
                   MAX_DISTANCE)

bounds = Polygon([(bMinLong, bMinLat),
                  (bMinLong, bMaxLat),
                  (bMaxLong, bMaxLat),
                  (bMaxLong, bMinLat),
                  (bMinLong, bMinLat)])

strip = shoreline.intersection(bounds)
vStrips.append(strip)
```

Затем обрабатываем каждую вертикальную полосу, нарезая ее на блоки размером с сегмент и сохраняя многоугольники внутри каждого блока в списке списков `tilePolys`:

```
stripNum = 0
for iLong in range(minLong, maxLong+1):
    vStrip = vStrips[stripNum]
    stripNum = stripNum + 1

    for iLat in range(minLat, maxLat+1):
        bMinLat,bMinLong,bMaxLat,bMaxLong = \
            expandRect(iLat, iLong, iLat+1, iLong+1,
                       MAX_DISTANCE)

        bounds = Polygon([(bMinLong, bMinLat),
```

```
                        (bMinLong, bMaxLat),
                        (bMaxLong, bMaxLat),
                        (bMaxLong, bMinLat),
                        (bMinLong, bMinLat)])

        polygon = vStrip.intersection(bounds)
        if not polygon.is_empty:
            tilePolys[iLat][iLong].append(polygon)
```

Сейчас мы сохранили нарезанные на сегменты многоугольники в `tilePolys` для каждого возможного значения `iLat/iLong`.

Теперь мы готовы снова сохранить (теперь уже) нарезанные на сегменты данные береговых линий в базе данных. Правда, прежде чем мы сможем это сделать, нам нужно создать новую таблицу базы данных, которая будет хранить нарезанные на сегменты многоугольники береговых линий. Вот необходимый программный код:

```
cursor.execute("DROP TABLE IF EXISTS tiled_shorelines")
cursor.execute("CREATE TABLE tiled_shorelines (" +
               " intLat INTEGER," +
               " intLong INTEGER," +
               " outline GEOMETRY(GEOMETRY, 4326)," +
               " PRIMARY KEY (intLat, intLong))")
cursor.execute("CREATE INDEX tiledShorelineIndex "+
               "ON tiled_shorelines USING GIST(outline)")
```

И наконец, мы можем сохранить нарезанные на сегменты многоугольники береговых линий в таблице `tiled_shorelines`:

```
for iLat in range(-90, +90):
    for iLong in range(-180, +180):
        polygons = tilePolys[iLat][iLong]
        if len(polygons) == 0:
            outline = Polygon()
        else:
            outline = shapely.ops.cascaded_union(polygons)
        wkt = shapely.wkt.dumps(outline)

        cursor.execute("INSERT INTO tiled_shorelines " +
                       "(intLat, intLong, outline) " +
                       "VALUES (%s, %s, " +
                       "ST_GeomFromText(%s))",
                       (iLat, iLong, wkt))
    connection.commit()

connection.close()
```

На этом наша программа нарезки береговых линий на сегменты завершена. Ее можно выполнить, набрав в окне терминала или командной строки следующую команду:

```
python3 tileShorelines.py
```

Отметим, что из-за объема обрабатываемых данных береговых линий на выполнение этой программы может потребоваться несколько часов. Ее можно запустить и оставить работать на ночь.

При первом выполнении программы вы, возможно, захотите поменять строку

```
for shoreline in shorelines:
```

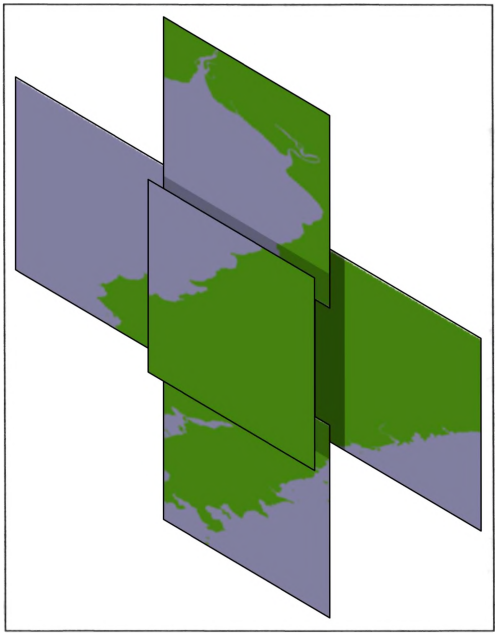
на следующую:

```
for shoreline in shorelines[1:2]:
```

В результате программа сможет завершить работу всего через пару минут, позволяя удостовериться, что она работает, как надо, прежде чем удалить `[1:2]` и выполнить ее на всей базе данных береговых линий.

Использование сегментов береговых линий

Вся проделанная работа в итоге дает нам новую таблицу базы данных, `tiled_shorelines` с данными береговых линий, нарезанными на частично накладывающиеся четырехугольные сегменты:



Поскольку все данные береговых линий для заданного набора поисковых результатов гарантированно будут находиться внутри одной записи таблицы `tiled_shoreline`, мы можем видоизменить наш сценарий `showResults.py`, с тем чтобы он использовал не исходные данные береговых линий, а нарезанные на сегменты.

Чтобы воспользоваться нарезанными на сегменты данными береговых линий, мы применим встроенную в источник данных `mapnik.PostGIS` умную функцию. Параметр `table`, передаваемый этому источнику данных, обычно представляет собой имя таблицы базы данных, содержащей данные для отображения. Однако наряду с обычным именем таблицы можно также передавать команду SQL `SELECT` для отбора конкретного подмножества записей для отображения. Мы воспользуемся ей, чтобы идентифицировать конкретную запись нарезанных на сегменты данных береговых линий для отображения.

К счастью, наша функция `mapGenerator.generateMap()` принимает параметр `tableName` с именем таблицы, который передается прямиком в источник данных `mapnik.PostGIS`. Это позволяет нам модифицировать передаваемое генератору карты имя таблицы, чтобы использовать подходящую запись нарезанных на сегменты данных береговых линий.

Для этого откройте в текстовом редакторе сценарий `showResults.py` и найдите оператор, который вызывает функцию `mapGenerator.generateMap()`. Этот оператор в настоящий момент выглядит следующим образом:

```
imgFile = mapGenerator.generateMap("shorelines",
                                   minLong, minLat,
                                   maxLong, maxLat,
                                   MAX_WIDTH, MAX_HEIGHT,
                                   points=points)
```

Замените этот оператор на приведенный ниже:

```
iLat = int(round(latitude))
iLong = int(round(longitude))

subSelect = "(SELECT outline FROM tiled_shorelines" \
          + " WHERE (intLat={})".format(iLat) \
          + " AND (intLong={})".format(iLong) \
          + ") AS shorelines"

imgFile = mapGenerator.generateMap(subSelect,
                                   minLong, minLat,
                                   maxLong, maxLat,
                                   MAX_WIDTH, MAX_HEIGHT,
                                   points=points)
```

С учетом этих изменений вместо полных данных береговых линий, полученных с веб-сайта географической базы данных GSHHG, сценарий `showResults.py` будет использовать данные береговых линий, нарезанные на сегменты. Теперь рассмотрим, какое повышение производительности нам дадут данные береговых линий в таком виде.

Анализ повышения производительности

Когда вы запустите новую версию приложения DISTAL, вы должны заметить улучшение производительности: теперь во время генерирования карты сценарий

showResults.py, похоже, возвращает свои результаты почти мгновенно, а не через пару секунд с небольшим. Добавление в сценарий showResults.py функционала хронометража показывает его текущее быстродействие:

```
Генерирование карты заняло 0.1074 сек.
```

Сравните это с тем, сколько времени требовалось, когда мы использовали полные данные береговых линий с высоким разрешением:

```
Генерирование карты заняло 3.0208 сек.
```

Как видите, мы получили драматическое улучшение производительности: генератор карты теперь в 15–20 раз быстрее, чем раньше, и суммарно затрачиваемое сценарием showResults.py время теперь меньше четверти секунды. Это неплохо для относительно простого изменения в наших базовых данных географической карты.

Заключение

В этой главе мы внесли ряд усовершенствований в приложение DISTAL, отрегулировали различные вопросы, связанные с удобством его использования и производительностью. По ходу мы узнали о линии антимеридиана и как поступать с контурами стран, которые простираются за ее пределы. Мы узнали, как добавить в наши сценарии CGI функционал масштабирования, так чтобы пользователь мог точно нажать кнопкой мыши на нужной точке поиска, и мы увидели, какое влияние на производительность нашей системы оказывает наличие огромных многоугольников. Затем мы научились нарезать эти большие многоугольники на меньшие по размеру перекрывающиеся четырехугольные сегменты, при помощи которых можно визуализировать небольшой участок береговых линий с высоким разрешением без последствий для производительности программы.

Теперь мы обладаем полнофункциональной версией системы DISTAL. Если потребуется, вы можете разместить систему DISTAL на публичном веб-сервере, сделав его доступным для любого, кто пожелает им воспользоваться. Правда, приложение DISTAL реализовано в виде серии сценариев CGI, то есть с использованием весьма рудиментарного метода реализации веб-приложения. В следующей главе мы рассмотрим географический модуль для веб-платформы Django под названием GeoDjango, который идеально подходит для разработки более технологичных геопространственных веб-приложений.

Глава 10

Инструменты для разработки геопространственных веб-приложений

В этой главе мы познакомимся с разнообразными инструментами и методами, при помощи которых можно разрабатывать геопространственные приложения, работающие в веб-браузере пользователя. Веб-приложения становятся все более популярными, и потенциал для разработки геопространственных приложений на основе этой технологии огромен.

Мы начнем наше изучение процесса разработки геоприложений, основанных на интернет-технологиях, обзором инструментария и методики, которые могут использоваться как для разработки веб-приложения в целом, так и для разработки геопространственных веб-приложений в частности. Мы рассмотрим веб-приложения, веб-службы, концепцию «скользящей карты» и несколько стандартных протоколов для совместного использования геоданных и управления ими через Интернет.

Затем мы обратимся к трем конкретным инструментам и методам, при помощи которых в трех заключительных главах этой книги будет разработано полнофункциональное геопространственное веб-приложение: протоколу TMS, картографической библиотеке OpenLayers на JavaScript и географическому модулю GeoDjango для веб-платформы Django.

Инструментарий и методика для геопространственных веб-приложений

Хотя изначально веб-браузеры были разработаны для вывода статических страниц HTML, теперь они по праву представляют собой технологичную среду программирования. Сложные приложения теперь можно реализовывать на JavaScript,

выполняющемся в веб-браузере пользователя, при взаимодействии с серверными API и веб-службами с выполнением задач, которые ранее можно было реализовать только при помощи сложных автономных систем.

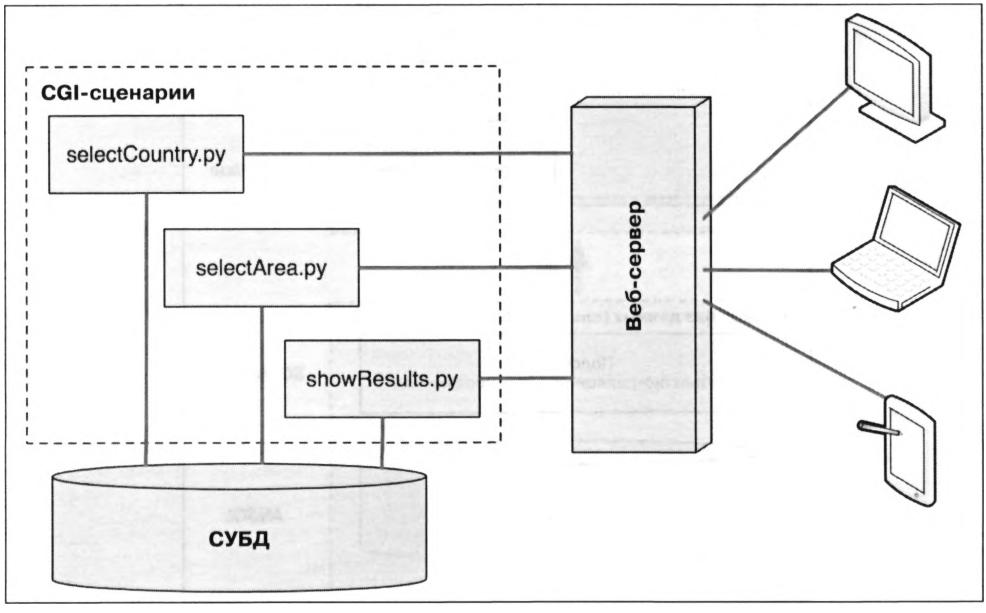
Процесс разработки веб-приложения трансформировал ландшафт программирования, и разработчики геоприложений не остались без внимания. В этом разделе мы познакомимся с процессом разработки веб-приложений и увидим, как эти интернет-технологии могут применяться для создания высокотехнологичных геопространственных систем, которые работают внутри веб-браузера пользователя.

Веб-приложения

Процесс разработки веб-приложения может осуществляться по-разному. Можно написать собственный программный код вручную, например в виде серии сценариев CGI, или же воспользоваться одной из многих доступных платформ для создания веб-приложений. В этом разделе мы рассмотрим несколько приемов структурирования веб-приложения таким образом, чтобы различные его части могли взаимодействовать между собой в целях реализации функциональности всего приложения.

Базовый подход

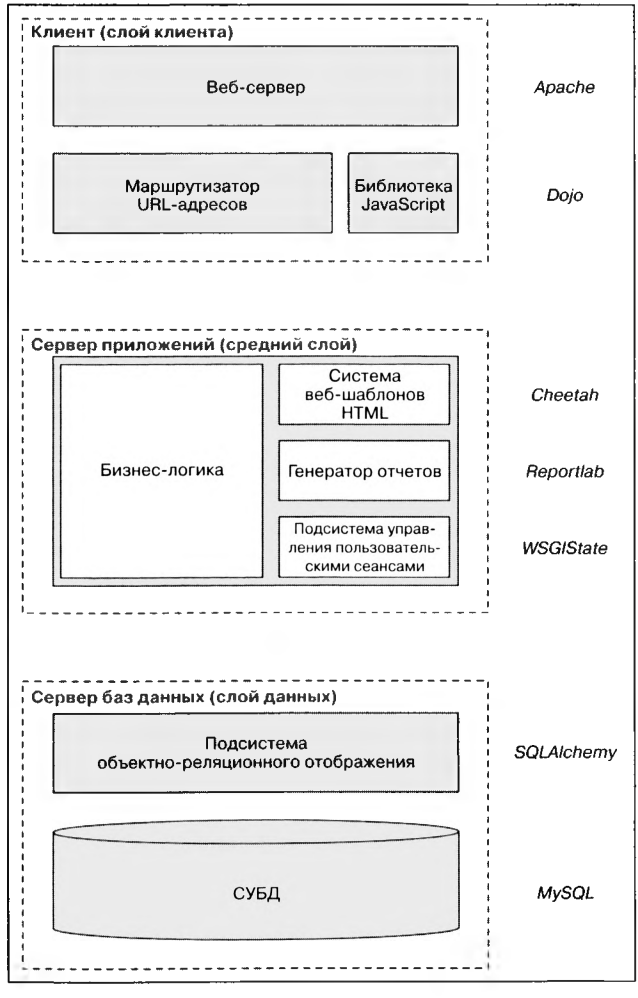
В *главе 8 «Работа с пространственными данными»* мы создали простое веб-приложение под названием DISTAL. Это веб-приложение было создано с использованием сценариев CGI и выполняло идентификацию городов и других объектов, исходя из расчетного расстояния. Приложение DISTAL – хороший пример базового подхода к разработке веб-приложения, в котором задействованы всего три составляющие: веб-сервер, база данных и набор сценариев CGI:



Преимущество такого подхода состоит в том, что для написания веб-приложения в таком виде вам не нужны специальные инструменты или знания. Недостаток же заключается в том, что приходится выполнять весь объем низкоуровневого программирования вручную. Описанный способ создания веб-приложения является утомительным и медленным, в особенности для сложного веб-приложения с широким спектром функциональных возможностей.

Стеки веб-приложений

Чтобы облегчить задачу создания веб-приложений, обычно используют существующие программные инструменты, которые позволяют писать приложение на более высоком уровне. Например, вы можете принять решение реализовать сложное веб-приложение следующим образом:



Инструменты приведенного выше стека взаимодействует между собой в целях реализации всего приложения. **Слой данных**, расположенный на самом низком уровне, занимается хранением данных. В приведенном примере веб-приложение использует MySQL в качестве СУБД и SQLAlchemy в качестве подсистемы объектно-реляционного отображения, предоставляющей объектно-ориентированный интерфейс к этой СУБД. **Средний слой** (уровень приложений) содержит бизнес-логику приложения, а также различные библиотеки, которые упрощают задачу создания сложного веб-приложения с отслеживанием состояний. Наконец, **слой клиента** (уровень представления) занимается пользовательским интерфейсом, раздавая веб-страницы пользователям, преобразуя поступающие URL-адреса в вызовы соответствующих функций вашей бизнес-логики и пользуясь тщательно продуманной библиотекой JavaScript для создания сложных пользовательских интерфейсов внутри веб-браузера пользователя.

Эти три слоя нередко обозначаются разными терминами. Например, слой данных иногда называют слоем доступа к данным, а слой приложений иногда упоминается как слой логики. Как бы то ни было, концепция остается прежней.

Отметим, что единственными компонентами этого приложения, которые разработчик создает с нуля, являются бизнес-логика, маршрутизация URL-адресов и схема базы данных, которая используется подсистемой объектно-реляционного отображения. Все остальное представляет собой простой набор стандартных компонентов.

Не стоит слишком зацикливаться на деталях архитектуры этого конкретного приложения – требуется всего лишь понять, что существует **стек** взаимодействующих между собой инструментов, в котором каждый инструмент использует инструменты, находящиеся уровнем ниже его. Кроме того, отметим сложность этой системы: это приложение зависит от *большого* количества различных инструментов и библиотек. Процессы разработки, развертывания и обновления такого приложения могут быть сопряжены с определенными трудностями из-за большого числа различных составных частей.

Платформы для разработки веб-приложений

Чтобы избежать сложностей, вызванных смешиванием и подгонкой большого числа разных составных частей, веб-разработчики создали всевозможные **платформы**, в которых инструменты комбинируются, с тем чтобы предложить полнофункциональную систему разработки веб-приложений. Вместо того чтобы с неизбежностью выбирать, устанавливать и разворачивать с десяток разных библиотек, можно просто выбрать полнокомплектную платформу, которая объединяет в себе хорошо известный набор библиотек и пополняет его собственной логикой, чтобы в итоге предложить полноценный опыт разработки веб-приложения по типу «все включено». Большинство подобных наборов инструментов предлагает вам встроенную логику для обработки таких задач, как:

- определение и мигрирование схемы базы данных;
- отслеживание сеансов и обработка аутентификации пользователя;

- создание технологичных пользовательских интерфейсов, часто с использованием AJAX для обработки сложных виджетов внутри браузера пользователя;
- автоматическое разрешение создания, чтения, обновления и удаления записей в базе данных пользователем (так называемый интерфейс CRUD);
- упрощение создания приложений под управлением базы данных посредством стандартных шаблонов и рецептов.

Этот список возможностей веб-платформ можно было бы продолжить, однако сейчас важно запомнить одно – они стремятся обеспечить «полностековый» функционал, чтобы позволить разработчикам быстро реализовывать наиболее распространенные аспекты веб-приложения с минимальной суетой. Они стремятся реализовать концепцию **быстрой разработки приложений**, англ. термин Rapid Application Development (**RAD**), в отношении систем, основанных на интернет-технологиях.

Ряд платформ для создания веб-приложений основан на языке Python, три из которых поддерживают конструирование геопространственных приложений: Django со встроенным в него географическим расширением GeoDjango, Pyramid с расширением Mapfish и Turbogears с расширением `tgext.geo`. Чуть позже в этой главе мы приглядимся поближе к географическому модулю GeoDjango.

Библиотеки пользовательского интерфейса

Несмотря на то что создать простой веб-интерфейс в HTML достаточно легко, пользователи все больше ожидают от веб-приложений, что они смогут конкурировать с настольными приложениями с точки зрения их пользовательского интерфейса. Такие операции, как выбор объектов, нажатие на них, рисование изображения мышью и перетаскивание объектов, больше не ограничены исключительно настольными приложениями.

Технология асинхронного JavaScript и XML, англ. термин Asynchronous Javascript And XML (**AJAX**), раньше, как правило, применялась в веб-приложениях для создания сложных пользовательских интерфейсов. В частности, выполнение кода JavaScript в веб-браузере пользователя наделяет приложение возможностью динамически отвечать на действия пользователя и заставляет веб-страницу вести себя таким образом, которого просто невозможно достичь при помощи статических страниц HTML.

Хотя язык JavaScript стал применяться повсеместно, временами возникают неожиданности, которые мешают писать на нем программы. Разные веб-браузеры, в которых может выполняться программный код на JavaScript, имеют свои собственные причуды и ограничения, затрудняя написание программного кода, который в любом браузере выполнялся бы одинаково. Кроме того, программный код на JavaScript очень низкоуровневый, требуя подробных манипуляций с содержимым веб-страницы в целях достижения заданного результата. Например, при реализации всплывающего меню требуется создание элемента `<DIV>`, который содержит нужным образом отформатированное (обычно при помощи CSS) и исходно

невидимое меню. Когда пользователь щелкает по соответствующей части страницы, всплывающее меню нужно показать, делая соответствующий элемент <DIV> видимым. Затем нужно отреагировать на манипуляции пользователем мышью по каждому элементу меню, визуально выделяя этот элемент и отменяя выделение ранее выделенного элемента. Затем, когда пользователь щелкает по элементу, необходимо снова скрыть меню, а потом ответить на действие пользователя.

Чтобы разобраться в таком подробном низкоуровневом программировании, могут потребоваться недели – в особенности имея дело с несколькими типами браузеров и их различными версиями. Поскольку вам в данном случае всего лишь нужно создать всплывающее меню, которое позволяет пользователю выбрать действие, то даже не стоит пробовать заниматься всем этим низкоуровневым программированием самостоятельно. Вместо этого, как правило, следует использовать одну из имеющихся **библиотек пользовательского интерфейса**, которая сделает всю тяжелую работу за вас.

Такие библиотеки пользовательского интерфейса написаны на JavaScript, и вы обычно добавляете их на свой веб-сайт, предоставляя доступ к файлу(ам) библиотеки JavaScript для скачивания и затем добавляя в вашу страницу HTML строку, с тем чтобы эту библиотеку можно было импортировать, как показано ниже:

```
<script type="text/javascript" src="library.js">
```

Если вы пишете свое собственное веб-приложение с нуля, то, чтобы реализовать пользовательский интерфейс для своего приложения, вы обратитесь к предназначенной для этого библиотеке. Более того, многие платформы для создания веб-приложений, в чей состав включена такая библиотека пользовательского интерфейса, напишут нужный код за вас, делая излишним даже этот шаг.

Вы можете предпочесть использовать в своем веб-приложении библиотеку пользовательского интерфейса, выбрав одну из десятка разных библиотек. Три наиболее популярных экземпляра представлены библиотеками JQuery UI, AngularJS и Bootstrap. При написании геопространственных веб-приложений легко забыть, что геопространственные веб-приложения – это прежде всего обычные веб-приложения, которые, помимо всего прочего, работают с геоданными. Подавляющая часть функциональности геопространственного веб-приложения довольно приземленная: обеспечение выдержанного оформления и функционала, реализация меню или панели инструментов для перехода между страницами, регистрация пользователя в системе, вход в систему и выход из нее, ввод обычных (не геопространственных) данных, создание отчетов и т. д. Вся эта функциональность может быть обработана одной из этих библиотек пользовательского интерфейса общего назначения, и вы свободны выбрать одну или несколько библиотек, исходя из своих предпочтений, либо использовать библиотеку пользовательского интерфейса (UI), встроенную в любую веб-платформу, на которой вы остановились.

В дополнение к библиотекам пользовательского интерфейса общего назначения есть и другие библиотеки, специально предназначенные для реализации геопространственных веб-приложений. Далее в этой главе мы изучим одну из таких библиотек – картографическую библиотеку OpenLayers.

Веб-службы

Веб-служба – это составная часть программного обеспечения, имеющая прикладной программный интерфейс (API), доступ к которому получают при помощи протокола HTTP. Веб-службы негласно реализуют функциональность, используемую другими системами; они обычно вообще не взаимодействуют с конечными пользователями.

К веб-службам получают доступ по URL-адресу; другие части системы отправляют на этот URL-адрес запрос наряду с другими параметрами и получают назад ответ, часто в форме данных в кодировке XML или JSON, которые затем используются для последующей обработки.

Имеются два главных типа веб-служб, с которыми вы, скорее всего, встретитесь: веб-службы на основе архитектуры RESTful[1], в которых используются части непосредственно самого URL-адреса, которые сообщают веб-службе, что делать, и «большие веб-службы«, в которых для связи с внешним миром, как правило, используется протокол SOAP.

Термин **REST**, сокр. от англ. Representational State Transfer, переводится как **«передача состояния представления»**. Для идентификации выданного запроса этот протокол использует подпути внутри URL-адреса. Например, чтобы вернуть информацию о клиенте, в веб-службе может использоваться следующий ниже URL-адрес:

```
http://myserver.com/webservice/customer/123
```

В этом примере элемент URL-адреса `customer` задает требующийся тип информации (то есть «ресурс»), а `123` – это внутренний идентификатор нужного клиента. Веб-службы RESTful легко реализовать и использовать, и поэтому они становятся все более популярными среди разработчиков веб-приложений.

«Большая веб-служба», с другой стороны, имеет всего один URL-адрес для всей веб-службы. На этот URL-адрес отправляется запрос, обычно в форме XML-сообщения, и в том же самом формате ответ передается обратно. Протокол SOAP нередко используется для описания формата сообщения и того, каким образом веб-служба должна себя вести. Большие веб-службы популярны в больших коммерческих системах, несмотря на то что являются сложнее своих собратьев RESTful.

Образец веб-службы

Рассмотрим простую, но полезную веб-службу. Приведенный ниже сценарий CGI под названием `greatCircleDistance.py` вычисляет и возвращает расстояние между двумя координатами на поверхности Земли, рассчитанное по дуге большого круга. Вот полный исходный код этой веб-службы:

---

[1] Термин RESTful применяют для веб-служб, построенных с учетом архитектуры REST. REST – это распространенная форма протоколов обмена сообщениями для взаимодействия компонентов распределенного приложения в сети. Прикладной программный интерфейс (API) REST определяет набор функций, к которым разработчики могут совершать запросы и получать ответы по протоколу HTTP. – *Прим. перев.*

```
#!/usr/bin/env python3

import cgi
import pyproj

form = cgi.FieldStorage()

lat1  = float(form['lat1'].value)
long1 = float(form['long1'].value)
lat2  = float(form['lat2'].value)
long2 = float(form['long2'].value)

geod = pyproj.Geod(ellps="WGS84")
angle1,angle2,distance = geod.inv(long1, lat1, long2, lat2)

print('Content-Type: text/plain')
print()
print("{:.4f}".format(distance))
```

Поскольку этот сценарий предназначается для использования прежде всего другими системами, а не конечными пользователями, обе координаты передаются как параметры запроса, а результирующее расстояние (в метрах) возвращается как тело HTTP-отклика. Поскольку возвращаемое значение представлено единственным числом, нет никакой потребности в кодировании результата при помощи XML или JSON; напротив, расстояние возвращается в виде простого текста.

Чтобы сделать программу расчета расстояния `greatCircleDistance.py` доступной для использования другими программами, можно воспользоваться сценарием веб-сервера `webserver.py`, который мы написали в *главе 8 «Работа с пространственными данными»*. Просто сделайте сценарий `greatCircleDistance.py` исполнимым и поместите его в своем каталоге `cgi-bin`, затем наберите `python3 webserver.py`, чтобы сделать вашу веб-службу доступной для других программ.

Теперь взглянем на простую программу Python, которая вызывает эту веб-службу:

```
import urllib

URL = "http://127.0.0.1:8000/cgi-bin/greatCircleDistance.py"

params = urllib.urlencode({'lat1' : 53.478948, # Манчестер.
                           'long1' : -2.246017,
                           'lat2' : 53.411142, # Ливерпуль.
                           'long2' : -2.977638})

params = params.encode("utf-8")
req = urllib.request.Request(URL)
with urllib.request.urlopen(req,data=params) as f:
    resp = f.read()
    print(resp.decode("utf-8"))
```

Выполнив эту программу, мы получим расстояние в метрах между двумя координатами, которые, кстати говоря, представляют географические положения городов Манчестер и Ливерпуль в Англии:

```
% python3 callWebService.py
49194.4632
```

Этот пример, может, показался не очень убедительным, но веб-службы – это чрезвычайно важная составная часть процесса разработки веб-приложений. Разрабатывая свои геопространственные веб-приложения, вы вполне можете пользоваться существующими веб-службами, а можете теоретически реализовать свои собственные, которые могут стать составной частью вашего веб-приложения.

Визуализация карт при помощи веб-службы

В *главе 8 «Работа с пространственными данными»* мы узнали, как при помощи библиотеки Mapnik можно генерировать привлекательные географические карты. В контексте веб-приложения визуализация цифровой карты обычно выполняется веб-службой, которая принимает запрос и возвращает визуализированную карту в виде графического файла. Например, ваше приложение может содержать средство визуализации цифровых карт по относительному URL-адресу `/render`, который принимает следующие параметры строки запроса:

Параметр(ы)	Описание
minX, maxX, minY, maxY	Минимальная и максимальная широта и долгота участка, содержащаяся в карте
width, height	Ширина и высота пиксела в сгенерированном изображении цифровой карты
layers	Разделенный запятыми список слоев, которые должны быть включены в состав карты. Предустановленные слои: «coastline» (береговая линия), «forest» (лесной покров), «waterways» (гидрография), «urban» (городские зоны) и «street» (дорожно-уличная сеть)
format	Нужный формат изображения. Один из следующих: «PNG», «JPEG» и «GIF»

Гипотетическая веб-служба `/render` вернет вызывающей стороне визуализированную карту в виде графического файла. После настройки веб-службы она будет действовать как черный ящик, предлагая изображения цифровых карт по запросу из других компонентов вашего веб-приложения.

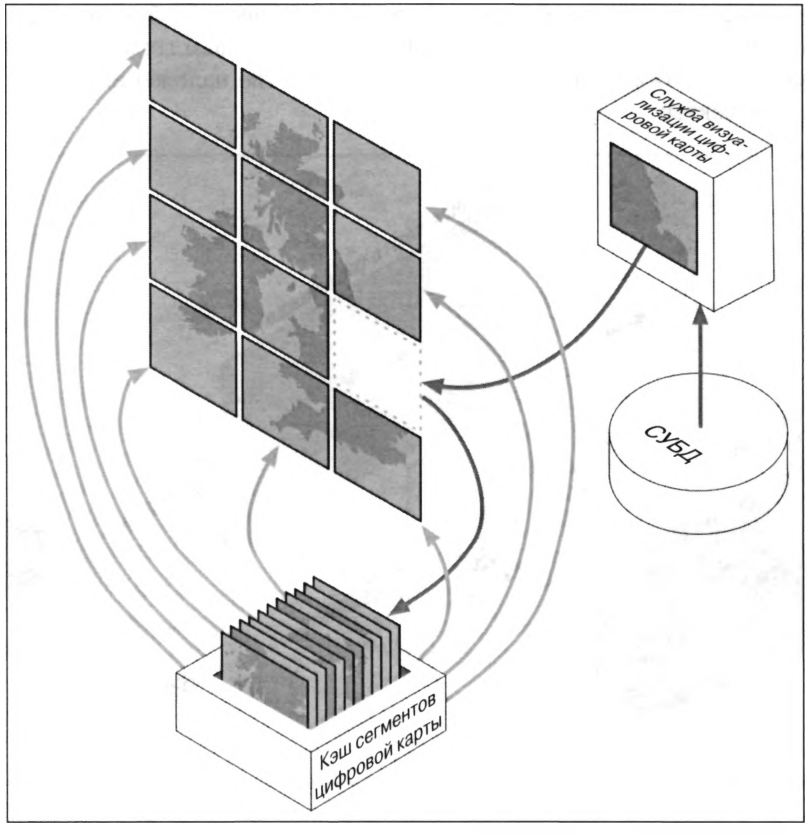
В качестве альтернативы размещению на своих ресурсах (хостинг) и конфигурированию вашего собственного средства визуализации цифровых карт можно решить использовать общедоступное внешнее средство визуализации. Например, веб-картографический проект OpenStreetMap предлагает свободное средство визуализации цифровых карт для данных OpenStreetMap на http://staticmap.openstreetmap.de.

Кэширование сегментов цифровой карты

С учетом того, что создание изображения из исходных данных географической карты является ресурсоемкой операцией как по количеству времени, так и по интенсивности работы процессора, ваше веб-приложение может в любой момент оказаться перегруженным, в случае если вы получите слишком много запросов на получение изображений цифровых карт. Как мы убедились, работая с приложением DISTAL в *главе 7 «Генерирование карт при помощи Python и библиотеки*

Mapnik», имеется масса, чего еще можно сделать, чтобы усовершенствовать быстродействие процесса генерирования карт, однако все еще остаются ограничения в отношении количества карт, которые приложение может визуализировать за определенный промежуток времени.

Поскольку данные географической карты обычно довольно статичны, в работе приложения можно сделать огромное улучшение, если предусмотреть **кэширование** сгенерированных изображений. Это обычно делается путем нарезки карты мира на **сегменты** с выводом изображений сегментов по мере необходимости и затем склейкой сегментов для получения целой карты:



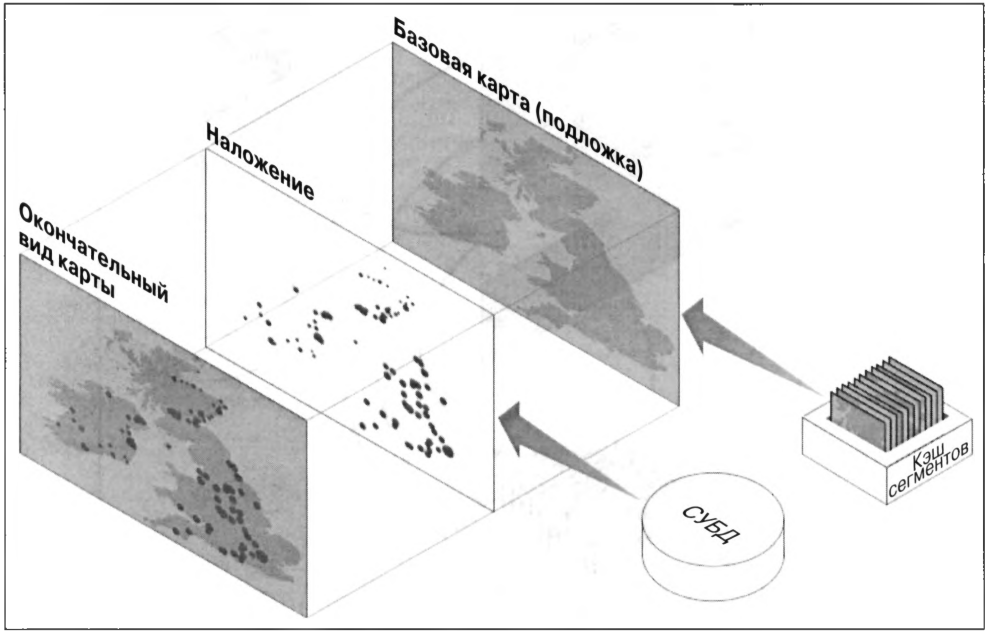
Кэш сегментов цифровой карты работает аналогично любому другому кэшу:

- когда сегмент запрашивается, кэш сегментов проверяется на наличие копии визуализированного сегмента. Если она имеется, то кэшированная копия сразу же возвращается;
- в противном случае вызывается служба визуализации цифровой карты, чтобы сгенерировать сегмент, и прежде чем только что сгенерированный сегмент будет возвращен вызывающей стороне, он добавляется в кэш;

- когда размер кэша становится слишком большим, сегменты, которые не были востребованы в течение долгого времени, удаляются, чтобы освободить место для новых сегментов.

Разумеется, кэширование сегментов будет работать, только если базовые данные географической карты не меняются. Как мы уже убедились, создавая приложение DISTAL, использовать кэш сегментов, когда визуализированное изображение варьируется от одного запроса к другому, не получится.

Одно из интересных применений кэша сегментов состоит в комбинировании их с **картографическими наложениями** (оверлеями) в целях улучшения работы даже в тех случаях, когда данные географической карты на самом деле изменяются. Учитывая, что контуры стран и других физических объектов на карте не меняются, при помощи генератора цифровых карт с кэшем сегментов можно сгенерировать «подложку», на которую затем в качестве наложения нанести изменяющиеся объекты:



Окончательную карту можно сгенерировать при помощи библиотеки Mapnik путем нанесения наложения на подложку, доступ к которому обеспечивается растровым источником данных `RasterDataSource` и которое отображается при помощи символизатора растра `RasterSymbolizer`. Если у вас достаточно дискового пространства, то вы даже можете предварительно вычислить все основные сегменты цифровой карты и иметь их в своем распоряжении для быстрого вывода. Такое использование библиотеки Mapnik представляет собой быстрый и эффективный способ объединить изменчивые данные географической карты с постоян-

ными в одной-единственной визуализации. Впрочем, есть и другие способы наложения данных на карту, например при помощи картографической библиотеки OpenLayers для вывода нескольких слоев карты одновременно.

Стэк «скользящей карты»

Концепция «скользящей карты» была популяризирована картографической веб-службой Карты Google. Там она называется масштабируемой (zoomable) картой, где пользователь может, нажав на карте и перетаскивая ее, прокручивать карту в разные стороны и двойным нажатием увеличивать ее масштаб. Вот образец скользящей карты картографической веб-службы Карты Google, которая показывает участок Европы:

Авторские права на изображение – Google;
авторские права на данные географической карты – Europa Technologies, PPWK и Tele Atlas

Скользящие карты стали чрезвычайно популярными, и подавляющая часть выполненной до сих пор работы по разработке геопространственного веб-приложения была сосредоточена именно на создании скользящих карт и работе с ними.

Интерфейс скользящей карты обычно реализуется при помощи специального программного стека, как показано ниже:

Начиная с основания, исходные данные географической карты обычно хранятся в пространственной базе данных. Затем они визуализируются при помощи такого инструмента, как библиотека Mapnik; для ускорения повторного доступа к тем же самым изображениям цифровой карты используется кэш сегментов. Затем берется библиотека пользовательского интерфейса, такая как OpenLayers, при помощи которой в веб-браузере пользователя выводится карта и возвращается ответ, когда пользователь нажимает на карте. Наконец, при помощи веб-сервера веб-браузеры получают возможность обращаться к скользящей карте и с ней взаимодействовать.

Геопространственные веб-протоколы

Учитывая то, что веб-приложения обычно подразделяются на несколько компонентов, способ обмена данными между компонентами становится чрезвычайно

важным. Вполне возможно, что ваше веб-приложение будет использовать готовые компоненты либо полагаться на уже существующие, работающие на удаленном сервере. В этих случаях крайне важны протоколы, которые используются для связи между разнообразными компонентами, позволяя этим разнообразным компонентам взаимодействовать между собой.

В отношении геопространственных веб-приложений был разработан ряд стандартных протоколов, позволяющих разнообразным компонентам обмениваться данными между собой. Приведем некоторые наиболее распространенные веб-протоколы, касающиеся разработки геопространственных приложений:

- протокол **веб-службы изображений веб-карт**, англ. термин Web Map Service (**WMS**), предлагает стандартный способ получения веб-службой запроса на генерирование карты и возврата изображения цифровой карты назад вызывающей стороне. Полная спецификация протокола WMS находится на странице http://www.opengeospatial.org/standards/wms;
- протокол **веб-службы сегментов веб-карт**, англ. термин Web Map Tile Service (**WMTS**), позволяет веб-службе предоставлять сегменты цифровой карты по запросу. Спецификация протокола WMTS находится на странице http://www.opengeospatial.org/standards/wmts;
- протокол **веб-службы сборных цифровых карт**, англ. термин Tile Map Service (**TMS**), представляет собой упрощенный протокол доставки сегментов цифровых карт по требованию. Дополнительная информация о протоколе TMS находится на странице http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification;
- протокол **веб-службы покрытий**, англ. термин Web Coverage Service (**WCS**), предоставляет пространственные покрытия по требованию. Пространственное покрытие – это многослойный набор данных, собранных по всему участку, например, с описанием уровней влажности почвы, высот или типов растительности. Спецификация протокола WCS находится на странице http://www.opengeospatial.org/standards/wcs;
- протокол **веб-службы географических объектов**, англ. термин Web Feature Service (**WFS**), хранит геопространственные объекты и позволяет вызывающей стороне выпускать запросы и управлять этими объектами. Во многих отношениях WFS служит в качестве пространственной базы данных, незаметно работающей позади веб-службы. Полная спецификация протокола WFS находится на странице http://www.opengeospatial.org/standards/wfs.

Многие из этих протоколов были разработаны Открытым геопространственным консорциумом, англ. термин Open Geospatial Consortium (OGC), который представляет собой международную организацию по стандартизации, разработавшую эти протоколы, с тем чтобы различные геопространственные системы и службы могли взаимодействовать.

Анализ трех конкретных инструментов

В заключительных трех главах этой книги мы создадим полнофункциональное технологичное геопространственное веб-приложение под названием **ShapeEditor** (редактор фигур картографических объектов). ShapeEditor строится поверх нескольких существующих технологий; прежде чем мы сможем приступить к его реализации, мы должны с ними познакомиться. В этом разделе мы изучим три ключевых инструмента, при помощи которых будет реализовано веб-приложение ShapeEditor: протокол веб-службы сборных цифровых карт TMS, картографическую библиотеку OpenLayers и географический модуль GeoDjango для веб-платформы Django.

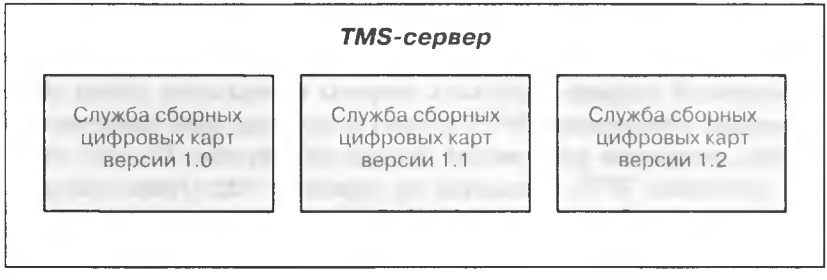
Протокол TMS

Протокол **веб-службы сборных цифровых карт TMS** описывает интерфейс веб-службы, которая возвращает изображения сегментов сборной цифровой карты по запросу. Протокол TMS аналогичен протоколу веб-службы изображений веб-карт WMS, за исключением того, что он проще и более ориентирован на хранение и извлечение сегментов, а не произвольно заданных полных карт.

Протокол TMS применяет принципы RESTful, а значит, используемый для доступа к веб-службе URL-адрес содержит всю информацию, необходимую для того, чтобы выполнить запрос. В отличие от протокола WMS, для того чтобы получить сегмент, отсутствует необходимость в создании и отправке сложных XML-документов – вся информация содержится внутри самого URL-адреса.

По своему содержанию протокол **TMS** представляет собой механизм для обеспечения доступа к визуализированным картам при наличии заданного набора масштабных коэффициентов и предварительно заданного набора систем пространственной привязки.

Один **сервер TMS** может принять несколько служб сборных цифровых карт:



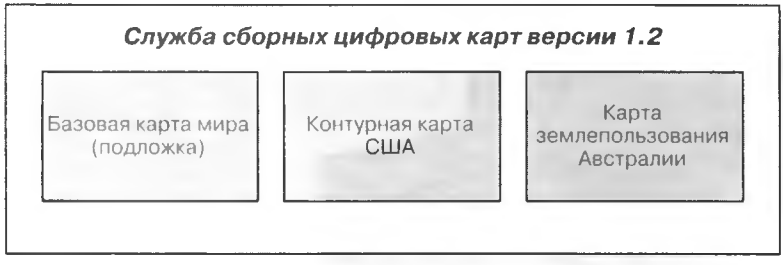
Такой подход, как правило, используется для того, чтобы предложить разные версии TMS-служб, реализуя его новые версии без вреда для клиентов, которые зависят от возможностей из более старой версии.

Каждая служба сборных цифровых карт внутри сервера TMS определяется по URL-адресу, который используется для получения доступа к конкретной службе.

Например, если сервер TMS работает на http://tms.myserver.com, то работающая на этом сервере версия 1.2 службы сборных цифровых карт обычно будет находиться по URL-адресу следующего уровня http://tms.myserver.eom/1.2/. Обращение к URL-адресу верхнего уровня (http://tms.myserver.com) вернет список всех имеющихся на этом сервере версий служб сборных цифровых карт:

```
<?xml version="1.0" encoding="UTF-8"/>
<Services>
  <TileMapService title="Служба TMS сервера MyServer" version="1.0"
    href="http://tms.myserver.com/1.0/"/>
  <TileMapService title="Служба TMS сервера MyServer" version="1.1"
    href="http://tms.myserver.com/1.1/"/>
  <TileMapService title="Служба TMS сервера MyServer" version="1.2"
    href="http://tms.myserver.com/1.2/"/>
</Services>
```

Каждая служба сборных цифровых карт обеспечивает доступ к одной или нескольким **сборным картам**:



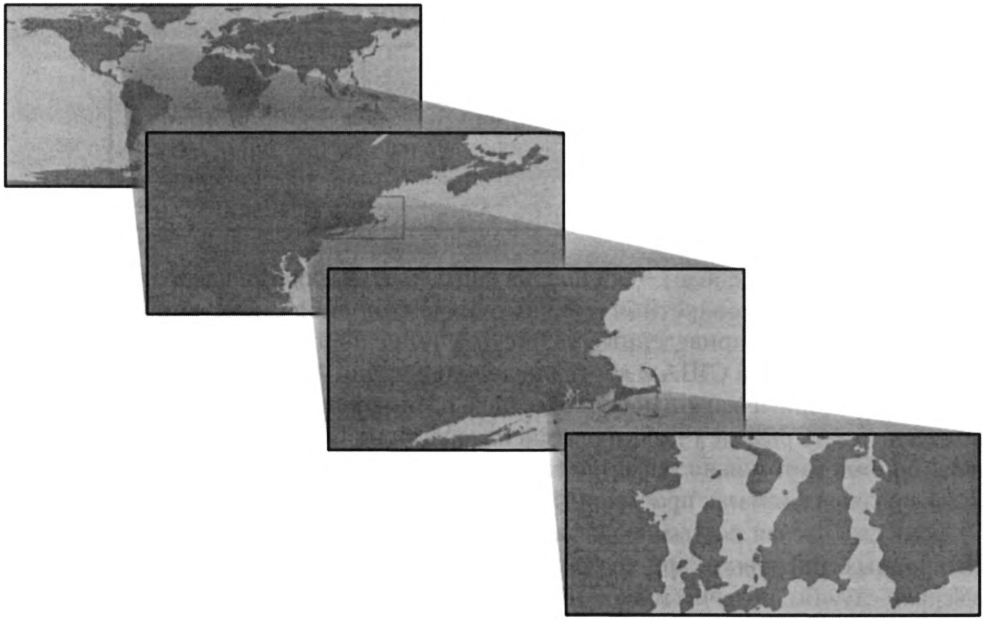
Сборная карта (tile map) – это полная карта всей Земли или какого-то ее участка, изображающая конкретные наборы геообъектов либо стилизованная особым образом. Примеры, приведенные на рисунке выше: базовая карта мира (подложка), контурная карта США и карта землепользования Австралии, — показывают, что разные сборные карты могут содержать разные виды географических данных или охватывать разные участки земной поверхности. Более того, при помощи разных сборных карт можно также доставлять карты в разных графических форматах либо в разных системах пространственной привязки.

Если клиентская система обращается к URL-адресу конкретной версии службы сборных цифровых карт, то последняя вернет более подробную информацию о версии службы, включая имеющийся в рамках этой службы список сборных карт:

```
<?xml version="1.0" encoding="UTF-8"/>
<TileMapService version="1.2" services="http://tms.myserver.com">
  <Title>Служба TMS сервера MyServer</Title>
  <Abstract>TMS-служба для сервера myserver.com</Abstract>
  <TileMaps>
    <TileMap title="Базовая карта мира"
             srs="EPSG:4326"
```

```
            profile="none"
            href="http://tms.myserver.com/1.2/baseMap"/>
  <TileMap title="Контурная карта США"
            srs="EPSG:4326"
            profile="none"
            href="http://tms.myserver.com/1.2/usaContours"/>
  <TileMap title="Карта землепользования Австралии"
            srs="EPSG:4326"
            profile="none"
            href="http://tms.myserver.com/1.2/ausLandUse"/>
  </TileMap>
</TileMaps>
</TileMapService>
```

Клиентским системам, обращающимся к визуализируемым картам через сервер TMS, обычно требуется возможность отобразить карту в различных разрешениях. Например, базовая карта мира первоначально может быть отображена как полная карта, и пользователь может увеличивать масштаб, чтобы получить более подробный вид нужного участка:



Этот процесс детализации изображения выполняется при помощи соответствующих **масштабных коэффициентов**. Каждая сборная карта состоит из нескольких подмножеств сегментов, или **листов** (tilesets), причем каждый лист

изображает карту с заданным масштабным коэффициентом. Например, первое изображение на приведенной выше иллюстрации было получено с масштабным коэффициентом приблизительно 1:100 млн, второе – 1:10 млн, третье – 1:1 млн, и последнее – с масштабным коэффициентом 1:100 тыс. Таким образом, внутри этой сборной карты имелись четыре листа, один для каждого масштабного коэффициента.

Если клиентская система обращается к URL-адресу заданной сборной карты, то сервер вернет об этой карте информацию, включая список имеющихся листов сборной карты:

```
<?xml version="1.0" encoding="UTF-8">
<TileMap version="1.2"
         tilemapservice="http://tms.myserver.com/1.2">
  <Title>Базовая карта мира</Title>
    <Abstract>Базовая карта, охватывающая весь мир</Abstract>
    <SRS>ESPG:4326</SRS>
    <BoundingBox minx="-180" miny="-90" maxx="180" maxy="90"/>
    <Origin x="-180" y="-90"/>
    <TileFormat width="256"
                height="256"
                mime-type="image/png"
                extension="png"/>
  <TileSets profile="none">
    <TileSet href="http://tms.myserver.com/1.2/basemap/0"
             units-per-pixel="0.703125"
             order="0"/>
    <TileSet href="http://tms.myserver.com/1.2/basemap/1"
             units-per-pixel="0.3515625"
             order="1"/>
    <TileSet href="http://tms.myserver.com/1.2/basemap/2"
             units-per-pixel="0.17578125"
             order="2"/>
    <TileSet href="http://tms.myserver.com/1.2/basemap/3"
             units-per-pixel="0.08789063"
             order="3"/>
  </TileSets>
</TileMap>
```

Отметим, что у каждого листа сборной карты есть свой собственный уникальный URL-адрес. При помощи этого URL-адреса будут получены отдельные **сегменты** внутри листа. Каждому сегменту присваивается значение прямоугольной координаты x и y, указывающей на его положение внутри всей карты. Например, используя предыдущую сборную карту, охватывающую всесь мир, третий лист будет состоять из 32 сегментов, расположенных в следующем порядке:

x = 0 y = 3	x = 1 y = 3	x = 2 y = 3	x = 3 y = 3	x = 4 y = 3	x = 5 y = 3	x = 6 y = 3	x = 7 y = 3
x = 0 y = 2	x = 1 y = 2	x = 2 y = 2	x = 3 y = 2	x = 4 y = 2	x = 5 y = 2	x = 6 y = 2	x = 7 y = 2
x = 0 y = 1	x = 1 y = 1	x = 2 y = 1	x = 3 y = 1	x = 4 y = 1	x = 5 y = 1	x = 6 y = 1	x = 7 y = 1
x = 0 y = 0	x = 1 y = 0	x = 2 y = 0	x = 3 y = 0	x = 4 y = 0	x = 5 y = 0	x = 6 y = 0	x = 7 y = 0

Такая конфигурация сегментов определяется следующей ниже информацией, полученной из сборной карты и выбранного листа:

- сборная карта использует систему пространственной привязки ESPG:4326, которая эквивалентна географическим координатам долготы и широты на основе датума WGS84. Иными словами, в данных географической карты используются значения географических координат широты и долготы, чьи значения долготы увеличиваются слева направо, а значения широты – снизу вверх;
- границы карты варьируются от −180 до +180 в направлении x (долгота) и от −90 до +90 в направлении y (широта);
- начало отсчета картографических координат находится в точке (−180, −90), то есть в нижнем левом углу карты;
- каждый сегмент сборной карты имеет 256 пикселов в ширину и 256 пикселов в высоту;
- третий лист имеет значение атрибута с числом единиц на пиксел `units-per-pixel`, равным 0.17578125.

Умножив значение числа единиц на пиксел на размер сегмента, мы увидим, что каждый сегмент охватывает 0.17578125×256 = 45° широты и долготы. Поскольку карта охватывает всю Землю, получим восемь сегментов по горизонтали и четыре сегмента по вертикали с началом отсчета координат в нижнем левом углу.

После того как клиентское программное обеспечение определилось с конкретным листом и рассчитало прямоугольные координаты x и y нужного сегмента, получение изображения сегмента просто сводится к конкатенации URL-адреса листа, координат x и y и расширения графического файла:

```
url = tileSetURL + "/" + x + "/" + y + "." + imgFormat
```

Например, чтобы получить сегмент в координате (3, 2) из приведенного выше листа, вы бы использовали следующий URL-адрес:

```
http://tms.myserver.com/1.2/basemap/2/3/2.png
```

Отметим, что этот URL-адрес выглядит (в действительности это касается всех используемых протоколом TMS URL-адресов), как будто вы просто получаете файл с сервера. На самом деле, чтобы сгенерировать эти сегменты по требованию, сервер TMS может незаметно выполнять серию сложных инструментов генерирования и кэширования карт – равно как и то, что весь сервер TMS может быть определен просто как серия статичных (неизменяемых) XML-файлов и несколько каталогов, содержащих предварительно сгенерированные графические файлы.

Понятие **статического сервера сборных цифровых карт** – преднамеренная конструктивная особенность протокола TMS. Если вам не приходится генерировать слишком много сегментов или вы располагаете особо емким жестким диском, то вы легко можете сгенерировать все изображения сегментов заранее и создать статический сервер TMS, приготовив несколько XML-файлов и раздавая данные, находясь позади стандартного веб-сервера, такого как Apache.

В последней главе этой книги мы создадим сервер TMS на основе Python в качестве компонента системы ShapeEditor. Вполне возможно, вы захотите использовать серверы TMS в собственных веб-приложениях и, вполне может быть, путем создания статического сервера сборных цифровых карт либо при помощи существующей программной библиотеки, которая реализует протокол TMS. Существуют две популярные библиотеки с открытым исходным кодом, которые предлагают инструменты кэширования сегментов сборной карты: TileCache (http://tilecache.org) и MapProxy (http://mapproxy.org), причем обе реализованы на Python.

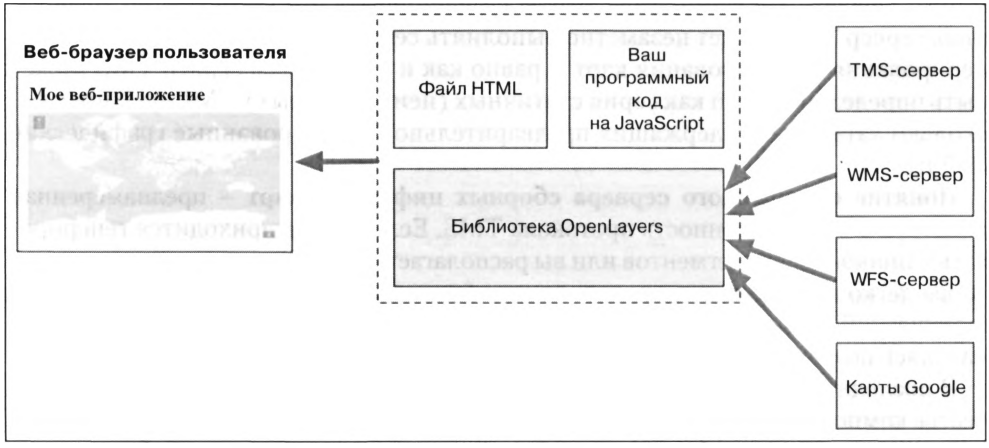
Библиотека OpenLayers

OpenLayers (http://openlayers.org) – это высокотехнологичная библиотека на JavaScript для создания картографических приложений. Она включает в себя прикладной программный интерфейс (API) JavaScript для создания скользящих карт, комбинирования данных из нескольких слоев и содержит различные виджеты для управления картами и для просмотра и редактирования векторных данных.

Версия 2 библиотеки OpenLayers существует уже много лет, тогда как полностью переработанная версия библиотеки под названием OpenLayers 3 появилась совсем недавно. Следует использовать библиотеку OpenLayers версии 3, поскольку в ней реализован намного более современный дизайн и она имеет активную поддержку. На момент написания настоящего текста последней была версия 3.10.1 данной библиотеки.

Чтобы воспользоваться библиотекой OpenLayers в своем веб-приложении, сначала необходимо создать файл HTML, который будет загружен в веб-браузер пользователя, и затем написать немного кода на JavaScript, в котором используется программный интерфейс библиотеки OpenLayers для построения нужной карты. Затем библиотека OpenLayers создает вашу карту и позволяет пользователю с ней взаимодействовать, загружая данные географической карты из определенного вами источника(ов) данных. Библиотека OpenLayers может читать из нескольких источников геоданных, в том числе серверов TMS, WMS и WFS. Все эти ком-

поненты взаимодействуют между собой, генерируя для вашего веб-приложения пользовательский интерфейс, как показано ниже:



Для работы с библиотекой OpenLayers необходимо уметь уверенно программировать на JavaScript. Это практически необходимость, когда вы создаете свои собственные веб-приложения. К счастью, программный интерфейс OpenLayers достаточно высокоуровневый и относительно упрощает процесс генерирования карт.

Ниже в качестве примера приведен HTML-код страницы, которая при помощи библиотеки OpenLayers выводит на экран скользящую карту:

```
<html>
  <head>
    <link rel="stylesheet"
          href="http://openlayers.org/en/v3.10.1/css/ol.css"
          type="text/css">
    <script src="http://openlayers.org/en/v3.10.1/build/ol.js"
            type="text/javascript">
    </script>
    <script type="text/javascript">
    function initMap() {
      var source = new ol.source.OSM();
      var layer = new ol.layer.Tile({source: source});
      var origin = ol.proj.fromLonLat([0, 0]);
      var view = new ol.View({center: origin, zoom: 1});
      var map = new ol.Map({target: "map",
                            layers: [layer],
                            view: view});
      }
```

```
    </script>
  </head>
  <body onload="initMap()">
    <div style="width:100%; height:100%" id="map"></div>
  </body>
</html>
```

Отметим следующую ниже строку:

```
<div style="width:100%; height:100%" id="map"></div>
```

Элемент будет содержать скользящую карту. Мы используем встроенный стиль, делая этот элемент во всю ширину и высоту страницы, а также назначаем идентификатору id значение «map».

Атрибут `onload = "initMap()"` в теге <body> страницы поручает веб-браузеру выполнить при загрузке страницы функцию JavaScript `initMap()`. Вся фактическая работа происходит внутри этой функции, которая определяет пять отдельных значений, при помощи которых создается скользящая карта:

- `source` – источник данных, который используется для получения базовых данных географической карты. В нашем примере мы выведем на экран данные проекта OpenStreetMap;
- `layer` – слой карты, который будет выведен на экран. В этом примере мы используем слой с нарезкой на сегменты для визуализации данных географической карты при помощи сегментов;
- `origin` – точка отсчета, на основе которой карта будет центрирована. Поскольку источник данных OpenStreetMap использует сферическую проекцию Меркатора, мы должны указать, чтобы библиотека OpenLayers при вычислении точки отсчета конвертировала нашу географическую координату (долготы и широты) в эту проекцию;
- `view` – участок карты, который будет отображаться при первой загрузке страницы. В этом случае карта будет центрирована на вычисленной точке отсчета и показана с максимальным масштабным уровнем;
- наконец, объект `map` содержит скользящую карту библиотеки OpenLayers. Значение `target` – это идентификатор элемента , в который карта будет выведена; помимо этого, мы назначаем карте вычисленные ранее объекты `layer` и `view`.

Если загрузить эту страницу HTML в свой веб-браузер, то вы должны увидеть, что скользящая карта показывает данные географической карты проекта OpenStreetMap:

Эта карта все еще довольно упрощенная, однако, используя различные объекты библиотеки OpenLayers, с ней можно сделать гораздо больше. Например, вы можете:

- отобразить несколько слоев карты, включая растровые изображения подложек и динамически генерируемые векторные данные;
- добавить на карту больше элементов управления;
- использовать *интерактивность*, определяющую, каким образом пользователь может управлять содержимым карты;
- позволить пользователю редактировать векторные данные географической карты непосредственно внутри карты.

Библиотека OpenLayers является мощным инструментом для создания геопространственных веб-интерфейсов. Даже если вы не используете ее непосредственно в своем собственном программном коде, многие платформы для создания веб-приложений, которые поддерживают разработку геопространственных веб-приложений (включая модуль GeoDjango для веб-платформы GeoDjango), используют библиотеку OpenLayers изнутри для визуализации и редактирования данных географической карты.

Модуль GeoDjango

GeoDjango – это расширение, встроенное в популярную веб-платформу Django (Джанго) для разработки веб-приложений[1]. Работая с GeoDjango, можно создавать сложные веб-приложения, которые позволяют пользователю просматривать и редактировать геоданные. Однако, прежде чем мы сможем приступить к работе с GeoDjango, мы должны понять принцип работы непосредственно самой веб-платформы Django.

Начнем с практического руководства, которое научит основам работы с веб-платформой Django. После того как вы разберетесь в основополагающих понятиях и создадите простое веб-приложение, мы рассмотрим, каким образом модуль GeoDjango опирается на веб-платформу Django, позволяя вам создавать свои собственные геопространственные веб-приложения. Здесь мы не будем вдаваться в слишком большое количество подробностей по поводу модуля GeoDjango, потому как мы будем активно работать с ним в заключительных трех главах этой книги.

Основы веб-платформы Django

Django (http://djangoproject.com) – это платформа **быстрой разработки приложений (RAD)** для создания веб-приложения на основе базы данных с использованием языка Python. Платформа Django весьма уважаема и используется в качестве движка для тысяч веб-приложений, развернутых в настоящее время по всей сети Интернет. Главные компоненты платформы Django включают в себя подсистему объектно-реляционного отображения, автоматически генерируемый веб-интерфейс администратора, гибкий маршрутизатор URL-адресов и систему шаблонной обработки. Все вместе эти компоненты Django позволяют быстро создавать технологичные веб-приложения для реализации большого спектра систем на основе базы данных.

В веб-платформе Django вы работаете с **проектом**, который состоит из нескольких отдельных приложений. Каждое приложение реализует набор определенного функционала, например аутентификацию пользователей, генерирование отчетов или ответы на API-запросы. Когда вы создаете проект, вы, как правило, пишете одно или несколько приложений самостоятельно и используете ряд предустановленных приложений, которые по умолчанию встроены в Django.

Одно из самых полезных встроенных приложений – **веб-интерфейс администратора**, который позволяет вам администрировать свое веб-приложение, просматривать и редактировать данные и т. д. Другие полезные встроенные приложения реализуют сеансы с сохранением состояния, аутентификацию пользователя, карты сайта, комментарии пользователей, рассылку электронной почты и постраничный просмотр данных. Кроме того, имеется большое количество приложений, дополненных пользователями.

---

[1] Технически GeoDjango представляет собой включаемый вспомогательный (contrib) модуль, который превращает Django в первоклассную географическую веб-платформу. – *Прим. перев.*

Воспользуемся веб-платформой Django, чтобы создать простое веб-приложение. Начнем с установки веб-платформы Django. Самый легкий способ подразумевает использование диспетчера пакетов Python pip. Из командной строки введите `pip install django`. Если он не работает, например, потому что у вас pip не установлен, то копию веб-платформы Django можно скачать на https://www.djangoproject.com/download.

После ее установки откройте окно терминала и при помощи команды `cd` перейдите в каталог, куда вы хотите поместить проект своего нового веб-приложения. Затем наберите `django-admin startproject example`.

В зависимости от того, как вы установили веб-платформу Django, программа `django-admin` может быть не доступна непосредственно из командной строки, потому что ее месторасположение может быть не прописано в системном пути. Если вы получаете ошибку, то вам, скорее всего, придется найти месторасположение установленной программы `django-admin` и дополнить свою команду соответствующим путем. Например, в Mac OS X команда будет такой: `/Library/Frameworks/Python.framework/Versions/current/bin/django-admin startproject example`. По желанию можно добавить каталог программы `django-admin` к системному пути, чтобы в следующие разы избежать необходимости набирать этот текст полностью.

В результате будет создан новый проект Django под названием `example`, который будет сохранен в каталоге с тем же самым именем. Этот каталог является просто контейнером для вашего проекта, в нем вы найдете следующее:

```
manage.py
example/
  __init__.py
  settings.py
  urls.py
  wsgi.py
```

Приглядимся поближе к этим файлам и папкам:

- `manage.py` – программа, которую Django генерирует автоматически. Вы будете пользоваться ею для выполнения и обновления своего проекта;
- подкаталог `example` – это пакет Python, который будет содержать различные файлы, связанные с вашим новым проектом. Как видите, он назван по имени вашего проекта;
- `__init__` – это пустой файл, который сообщает интерпретатору Python, чтобы он рассматривал подкаталог `example` в качестве пакета Python;
- модуль `settings.py` содержит различные настройки на уровне проекта;
- сценарий `wsgi.py` используется для выполнения вашего проекта внутри WSGI-совместимого веб-сервера.

Располагая проектом Django, теперь создадим простое приложение, которое будет работать внутри него. Наше приложение, которое мы назовем `hello`, просто выводит пользователю следующее приветственное сообщение:

Привет, Число X.

Здесь X – это число, которое увеличивается всякий раз, когда страница выводится на экран компьютера. Этот пример приложения может показаться тривиальным, вместе с тем он демонстрирует ряд ключевых функциональных компонентов веб-платформы Django:

- маршрутизация URL-адресов ресурсов в сети;
- режимы просмотра данных (views);
- шаблоны HTML;
- объектно-реляционное отображение.

Чтобы создать это приложение, перейдите при помощи команды `cd` во внешний каталог `example` и наберите следующую команду:

```
python3 manage.py startapp hello
```

В результате внутри внешнего каталога `example` будет создан новый подкаталог с именем `hello`. Этот каталог представляет собой пакет Python, который содержит различные части вашего нового приложения веб-платформы Django. Если вы заглянете в этот каталог, то найдете следующие элементы:

- `__init__.py` – еще один файл инициализации пакета, сообщающий интерпретатору Python, чтобы он рассматривал каталог `hello` как пакет Python;
- `admin.py` – модуль, который можно использовать для того, чтобы задать административный веб-интерфейс вашего приложения. Мы не будем им пользоваться, и поэтому его можно проигнорировать;
- каталог `migrations` содержит различные **миграции базы данных** для этого приложения. Содержимое этого каталога будет создано автоматически сценарием `manage.py`, когда мы определим структуру нашей базы данных;
- модуль `models.py` содержит различные **модели базы данных**, используемые этим приложением. Модель базы данных – это класс Python, который определяет структуру одной таблицы базы данных. Мы исследуем этот класс подробнее чуть позже;
- модуль `tests.py` – место, где определяются различные модульные тесты для вашего приложения;
- наконец, модуль `views.py` будет содержать различные **просмотровые функции**, которые реализуют поведение вашего приложения. Мы займемся редактированием этого модуля чуть позже.

Наша следующая задача состоит в том, чтобы сообщить проекту `example`, чтобы он использовал наше недавно созданное приложение `hello`. Для этого откройте в текстовом редакторе файл `example/settings.py` и отыщите переменную `INSTALLED_APPS`. Она содержит список используемых проектом приложений. Как видите, ряд встроенных приложений установлен по умолчанию. Отредактируйте этот список так, чтобы он был похож на приведенный ниже:

```
INSTALLED_APPS = (
    #'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'hello'
)
```

Отметим, что мы закомментировали приложение `django.contrib.admin`, поскольку для нашего простого проекта `example` веб-интерфейс администратора не понадобится. Мы также добавили в наш проект приложение `hello`.

На следующем этапе нужно установить модель базы данных. Поскольку наше приложение `hello` должно отслеживать количество появлений приветственного сообщения *Hello* на экране компьютера, мы создадим модель под названием `Counter` (Счетчик), которая как раз этим и займется. Для этого откройте свой модуль `hello/models.py` в текстовом редакторе и добавьте следующие строки в конце модуля:

```
class Counter(models.Model):
    count = models.IntegerField()
```

Модель базы данных (то есть подкласс класса `django.db.models.Model`) – это класс Python, который задает структуру таблицы базы данных. В нашем примере мы создали очень простую модель базы данных – всего одна модель `Counter`, которая имеет единственное поле `count`. С моделями баз данных еще много чего можно сделать, включая определение связей между моделями, настройку поведения по умолчанию, добавление в модели пользовательских методов и т. д., но этого для нашего проекта `example` будет достаточно.

Сохраните файл `models.py`, затем откройте окно терминала или командной строки, при помощи команды `cd` перейдите во внешний каталог `example` и наберите следующую ниже команду:

```
python3 manage.py makemigrations hello
```

Команда `makemigrations` проанализирует определенные вами для приложения `hello` модели базы данных и выяснит, какие произошли изменения со структурой базы данных с прошлого раза, когда команда была выполнена. Так как эта команда выполняется впервые, то будет создана новая таблица базы данных, которая соответствует структуре вашего класса `Counter`.

Если взглянуть вовнутрь каталога `hello/migrations`, то вы увидите, что был создан новый файл: `0001_initial.py`. Этот файл содержит подробные инструкции, необходимые веб-платформе Django, чтобы создать таблицу базы данных, соответствующую вашей модели базы данных.

После создания файла миграции нам нужно его применить. Для этого наберите следующую ниже команду:

```
python3 manage.py migrate
```

Команда `migrate` применяет все миграции базы данных, оставшиеся без применения. Так как в ваш проект включен ряд стандартных приложений, у них будут свои собственные миграции базы данных, которые будут применены в то же самое время.

Располагая базой данных, которая среди прочего содержит нашу новую модель Counter, резонно задастся вопросом, а где эта база данных находится. По умолчанию веб-платформа Django использует движок СУБД SQLite, создавая простую дисковую базу данных, которую она применяет для хранения данных вашего проекта. По желанию это поведение можно изменить, отредактировав настройки переменной DATABASES в вашем модуле example/settings.py.

Теперь, когда у нас есть база данных, давайте ею воспользуемся. Поскольку наш проект example должен выводить пользователю сообщение, на следующем этапе мы должны определить две вещи: **маршрутизацию URL-адресов**, которая сообщает Django, какие функции вызывать, когда пользователь пытается получить доступ к системе, и **просмотровую функцию**, которая выдаст ответ при обращении к URL-адресу, к которому она приписана.

В реальной системе эти преобразования URL-адресов могут становиться довольно хитроумными, но для нашей программы мы воспользуемся очень простым преобразованием. Откройте в текстовом редакторе модуль example/urls.py и добавьте к его заголовку следующий ниже оператор импорта:

```
from hello import views
```

Он сделает доступными относящиеся к приложению hello просмотровые функции (которые мы создадим в модуле hello/views.py) для использования в наших преобразованиях URL-адресов. Затем поменяйте существующее определение переменной urlpatterns нижеследующим:

```
urlpatterns = [
    url('^$', views.say_hello)
]
```

Отметим, что мы удалили ссылку на веб-интерфейс администратора, так как в нашем проекте мы его не используем. Функция url() определяет одно преобразование URL-адреса, которое ставит довольно загадочно выглядящий шаблон '^$' для URL-адреса в соответствие функции say_hello() внутри нашего модуля hello/views.py. Последовательность символов '^$' – это регулярное выражение, которое для целей нашего проекта соответствует URL-адресу верхнего уровня. Другими словами, если пользователь получит доступ к URL-адресу верхнего уровня, то будет вызвана функция views.say_hello(), чтобы выдать ответ на запрос пользователя.

Сохраните изменения, которые вы внесли в модуль urls.py, и затем откройте в текстовом редакторе модуль hello/views.py. Это то место, где вы определите просмотровые функции, относящиеся к приложению hello. Мы создадим там одну-единственную функцию под названием say_hello(), которая выдаст ответ, когда пользователь получит доступ к URL-адресу верхнего уровня нашего проекта.

Прежде чем мы напишем нашу функцию say_hello(), задумаемся на мгновение, что именно эта функция должна делать. Нам нужно вывести сообщение, которое выглядит следующим образом:

Привет, Число X.

Значение для *X* взято из нашей модели базы данных `Counter` и приращается всякий раз, когда вызывается наша просмотровая функция. Поэтому наш первый шаг должен загрузить запись `Counter`, создав ее, если она еще не существует, и затем увеличить ее на единицу. После этого на основе текущего значения счетчика мы можем вывести приветственное сообщение **Привет, Число X**.

Начнем с импорта класса `Counter` в наш модуль `views.py`:

```
from hello.models import Counter
```

Затем мы определяем нашу функцию `say_hello()`. В Django просмотровая функция принимает объект **запроса** (request), который предоставляет информацию об HTTP-запросе, который был получен от веб-браузера пользователя, и возвращает объект **ответа** (response), содержащий информацию, которая передается обратно в веб-браузер пользователя для отображения. Это означает, что базовая структура просмотровой функции веб-платформы Django выглядит следующим образом:

```
def view_function(запрос):
    ...
    return ответ
```

В данном случае нас на самом деле не интересует объект запроса, так как нам не нужна никакая информация, которую он может содержать. И напротив, нам действительно нужно вернуть ответ.

Начнем с определения первой части нашей просмотровой функции. Для этого добавьте следующий ниже оператор в конец своего модуля `views.py`:

```
def say_hello(request):
```

Наша просмотровая функция должна загрузить запись `Counter` из базы данных, при необходимости создав ее. Применив объектно-реляционное отображение веб-платформы Django, мы можем сделать это, используя класс `Counter` напрямую:

```
counter = Counter.objects.first()
if counter == None:
    counter = Counter(count=0)
```

Затем мы увеличиваем счетчик на единицу и сохраняем его назад в базу данных:

```
counter.count = counter.count + 1
counter.save()
```

Теперь у нас имеется значение для *X*, которое мы хотим включить в приветственное сообщение **Привет, Число X**. На заключительном этапе нам нужно создать объект HTTP-отклика, который содержит это сообщение, чтобы мы могли использовать его в качестве возвращаемого значения функции.

Хотя создать объект HTTP-отклика можно самыми разными способами, мы воспользуемся для этого встроенной в веб-платформу Django системой шаблонной обработки. Это легко сделать:

```
return render(request, "say_hello.html",
              {'count' : counter.count})
```

Функция `render()` принимает объект HTTP-запроса, имя используемого файла шаблона и «контекстный» словарь, содержащий значения, которые будут включены в шаблон. В данном случае мы воспользуемся шаблоном `say_hello.html` и предоставим значение `count`, которое будет включено в шаблон.

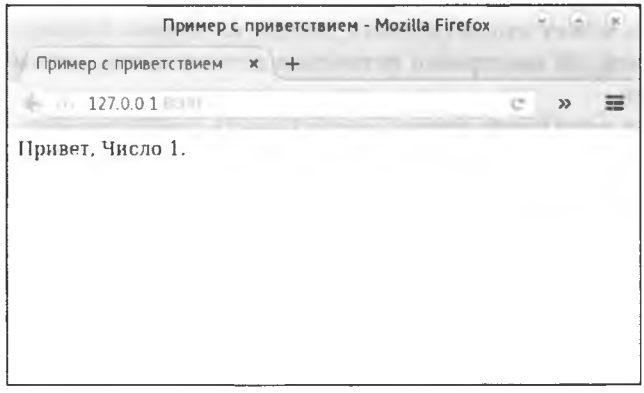
Мы почти завершили наш проект `example`. Осталось лишь создать HTML-шаблон: в каталоге приложения `hello` создайте новый каталог `templates` и затем создайте в нем файл с именем `say_hello.html`. Затем введите в этот файл следующий HTML-шаблон:

```
<html>
  <head>
    <title>Пример с приветствием</title>
  </head>
  <body>
    Привет, Число {{ count }}.
  </body>
</html>
```

Теперь можно выполнить проект, набрав в командной строке нижеследующее:

```
python3 manage.py runserver
```

Чтобы протестировать вашу программу, откройте веб-браузер и перейдите по адресу `http://127.0.0.1:8000`. Если все выполнено правильно, то вы должны увидеть, как в веб-браузере появится следующее сообщение:



Перезагрузка страницы будет каждый раз увеличивать значение счетчика.

Конечно, проект `example` довольно простой – многие подробности мы пропустили, и получившаяся в результате программа еще не выглядит достаточно хорошей, – но мы охватили большинство ключевых функциональных особенностей веб-платформы Django, о которых необходимо знать:

- как создавать проекты и приложения;
- как определять и использовать модели базы данных;
- как работают миграции базы данных;

- как работает маршрутизация URL-адресов, которая ставит просмотровые функции в соответствие URL-адресам;
- как написать простую просмотровую функцию;
- как создавать и использовать шаблон HTML.

Познакомившись с тем, как работает веб-платформа Django, теперь выполним краткий обзор того, как географический модуль GeoDjango расширяет веб-платформу Django, чтобы дать вам возможность создавать геопространственные веб-приложения.

Модуль GeoDjango

Модуль GeoDjango опирается на возможности веб-платформы Django в добавлении полной поддержки создания геопространственных веб-приложений. В частности, модуль дополняет веб-платформу Django следующим функционалом.

- Модель:
  - модель Django расширяется за счет предоставления возможности хранения и получения геоданных;
  - объектно-реляционное отображение (OR-mapping, ORM) Django расширяется за счет поддержки пространственных запросов;
  - по мере чтения геообъектов из базы данных объектно-реляционный преобразователь автоматически переводит их в объекты GEOS, предоставляя методы для выполнения запросов к этим объектам и управления ими самыми разными способами, аналогичными интерфейсу, предоставляемому библиотекой `Shapely`;
  - модель может импортировать данные из любого поддерживаемого библиотекой `OGR` векторного источника данных в базу данных модуля GeoDjango;
  - модуль GeoDjango может задействовать *интроспекцию*, помогая разобраться в том, какие атрибуты доступны в заданном источнике данных библиотеки `OGR`, и автоматически настраивать модель для сохранения и импорта этих атрибутов.
- Шаблон:
  - система шаблонной обработки Django расширяется за счет предоставления возможности визуализации геоданных, используя встроенную скользящую карту библиотеки OpenLayers.
- Веб-интерфейс администратора:
  - веб-интерфейс администратора Django расширяется, предоставляя пользователю возможности создавать и редактировать геоданные при помощи библиотеки OpenLayers. Векторные данные выводятся поверх подложки, предоставленной проектом OpenStreetMap.
- Калькуляторы расстояния и площади:
  - пакет `django.contrib.gis.measure` добавляет поддержку расчетов расстояний и площадей и работы с ними;
  - расстояния и площади могут конвертироваться в целый ряд стандартных единиц измерения, например в миллиметры, ярды или мили.

- IP-геолокация:
  - модуль GeoDjango содержит обертку вокруг программного интерфейса библиотеки MaxMind GeoiP[1], позволяя вычислять географические положения на основе IP-адреса вызывающей стороны;
  - местоположение можно получить по географической координате широты и долготы либо по названию города и страны.

В общем и целом модуль GeoDjango делает веб-платформу Django оптимальным выбором для разработки геопространственных веб-приложений. Более того, полнофункциональное геоприложение ShapeEditor, которое мы напишем в оставшейся части этой книги, будет фактически создано поверх веб-платформы Django и ее модуля GeoDjango.

Заключение

В этой главе мы узнали о разнообразных инструментах и методах, связанных с разработкой геоприложений, к которым можно обращаться через веб-интерфейс. Мы увидели, каким образом веб-приложения могут быть структурированы, узнали, что платформы для создания веб-приложений могут упростить процесс создания веб-приложения, и увидели, как библиотеки пользовательского интерфейса намного облегчают работу по реализации веб-приложений. Затем мы рассмотрели понятие веб-служб и научились реализовывать калькулятор расстояния по дуге большого круга как веб-службу. Мы также рассмотрели, каким образом можно реализовать визуализацию цифровой карты как веб-службу, и увидели, как кэширование сегментов может ускорить процесс отображения географических карт внутри веб-браузера.

Затем мы рассмотрели концепцию скользящих карт и увидели, как они создаются при помощи стека стандартных компонентов. Мы также исследовали несколько общих протоколов совместного использования и управления геоданными.

Наконец, мы всесторонне проанализировали три конкретных инструмента, которыми воспользуемся в оставшейся части этой книги: протокол веб-службы сборных цифровых карт TMS, библиотеку пользовательского интерфейса OpenLayers и модуль GeoDjango.

В следующей главе мы начнем создавать полнофункциональное картографическое приложение, воспользовавшись для этого расширением PostGIS, библиотекой Mapnik и модулем GeoDjango.

---

[1] Продукты MaxMind GeoiP идентифицируют местоположение и другие характеристики интернет-пользователей для различных областей применения, включая персонализацию контента, прицельное использование рекламы, анализ трафика, соответствие требованиям, геотаргетинг, установку геозон и управление авторскими правами. – *По материалам источника* https://www.maxmind.com/ru/geoip2-services-and-databases.

Глава 11

Собираем все вместе – полнофункциональная картографическая система

В трех заключительных главах этой книги мы сведем вместе все темы, которые обсуждались в предыдущих главах, чтобы реализовать полнофункциональное картографическое веб-приложение под названием ShapeEditor.

В этой главе мы:

- рассмотрим систему ShapeEditor с точки зрения пользователя, чтобы понять, как она будет работать;
- подробно проанализируем различные компоненты системы ShapeEditor и рассмотрим их реализацию с точки зрения структур данных и функциональности;
- настроим используемую системой ShapeEditor базу данных PostGIS;
- создадим для системы ShapeEditor проект и приложения GeoDjango;
- определим модели базы данных системы ShapeEditor;
- сконфигурируем для системы ShapeEditor веб-интерфейс администратора GeoDjango;
- воспользуемся веб-интерфейсом администратора для просмотра и редактирования геоданных в базе данных системы ShapeEditor

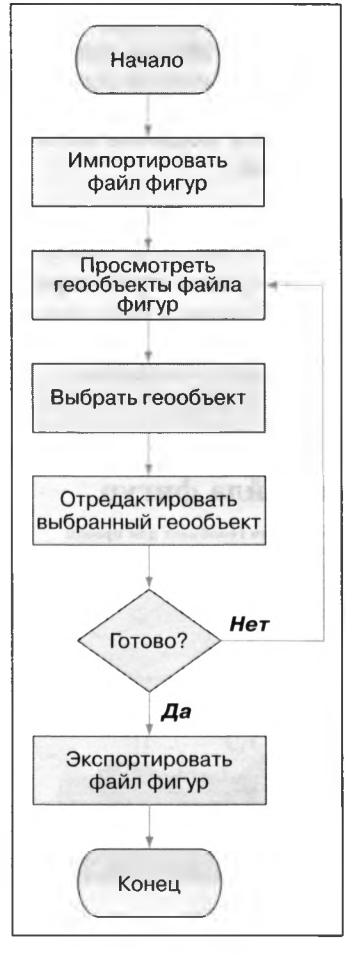
О системе ShapeEditor

Как мы убедились ранее, файлы фигур обычно используются для хранения, предоставления и передачи геоданных. В этой книге мы активно поработали с файлами фигур, при этом мы получали в этом формате общедоступные геродаанные, писали программы загрузки данных из таких файлов и создавали их программным путем.

В то время как редактирование связанных с геообъектами файла фигур атрибутов выполняется достаточно легко, правка геообъектов в них оказывается намного сложнее. Один из подходов состоит в инсталляции системы ГИС и ее использовании для импорта данных, внесении в них изменений и затем экспорта данных в другой файл фигур. Хотя этот подход и работает, его удобство сомнительно, если вы хотите внести в геообъекты файла фигур всего лишь несколько изменений. Было бы намного легче, если бы у нас было веб-приложение, специально разработанное для редактирования файлов фигур.

И это именно то, чем мы займемся: реализуем веб-ориентированный редактор файлов фигур, который, не мудрствуя лукаво, мы назовем редактором фигур ShapeEditor.

Следующая ниже блок-схема изображает основную последовательность операций системы ShapeEditor:

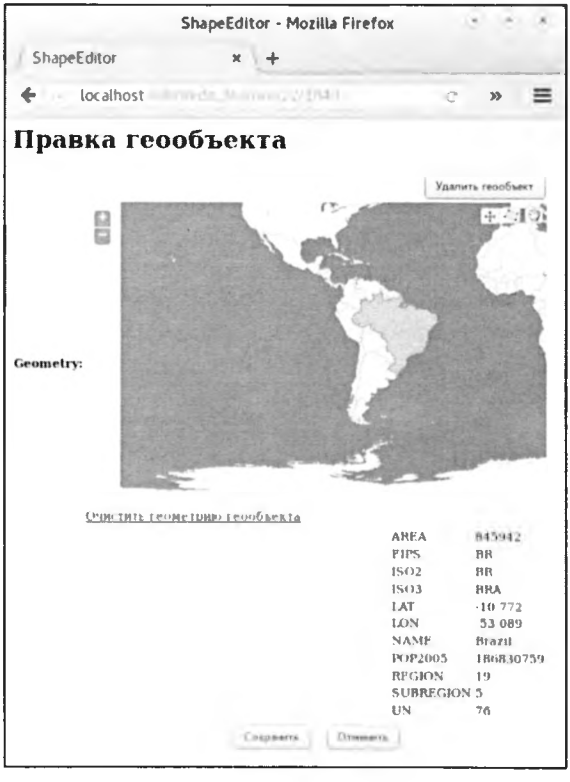


Пользователь начинает работу с импорта файла фигур, используя для этого веб-интерфейс ShapeEditor, как показано на следующем ниже снимке экрана:

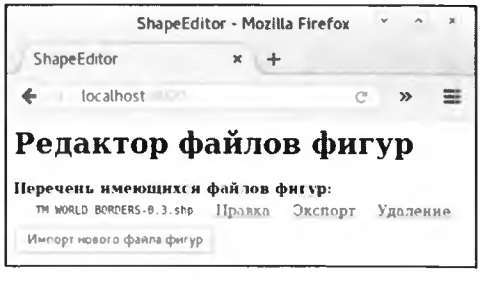
Наша реализация системы ShapeEditor была выбрана не для того, чтобы наделить ее привлекательным внешним видом; как раз напротив, она сконцентрирована на надежно работающем функционале. Чтобы улучшить внешний вид приложения, можно легко добавить таблицы стилей и отредактировать шаблоны HTML, но эта работа сделает программный код тяжелее и, соответственно, труднее для понимания. Именно поэтому мы приняли такой минималистский подход к пользовательскому интерфейсу. Создание его симпатичной версии оставлено читателю в качестве упражнения.

После того как файл фигур импортирован, пользователь может просмотреть геообъекты файла фигур на карте и, нажав на объекте, его выбрать. В этом случае мы импортировали набор данных границ стран мира World Borders Dataset, который мы уже несколько раз использовали в этой книге:

Далее пользователь может отредактировать геометрию выбранного геообъекта, а также увидеть список атрибутов, связанных с этим геообъектом:



После того как пользователь закончил вносить изменения в файл фигур, он или она может экспортировать файл фигур, нажав на гиперссылку **Экспорт** на главной странице:



Все это достаточно полно описывает функциональность ShapeEditor. Эта система сравнительно прямолинейная, но вместе с тем она может быть очень полезной, если вам нужно работать с геоданными в формате файла фигур. И конечно,

по ходу реализации ShapeEditor вы научитесь реализовывать свои собственные технологичные геопространственные веб-приложения, используя для этого географический модуль GeoDjango.

Проектирование системы ShapeEditor

Приглядимся поближе к различным компонентам системы ShapeEditor, чтобы увидеть, что будет задействовано при ее реализации. Система ShapeEditor будет поддерживать следующие ниже операции:

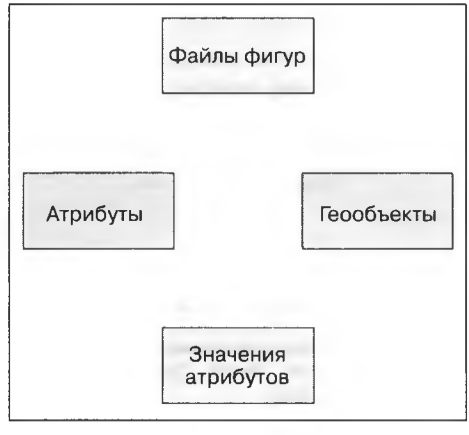
- импорт геопространственных объектов и атрибутов из файла фигур;
- предоставление пользователю возможности выбрать геообъект для правки;
- отображение подходящего типа редактора, чтобы дать пользователю отредактировать геометрию геообъекта;
- экспорт геопространственных объектов и атрибутов назад в файл фигур.

Приглядимся поближе к каждой из этих операций пользователя, чтобы понять, как их реализовать в рамках системы ShapeEditor.

Импорт файла фигур

Когда пользователь импортирует файл фигур, мы сохраняем содержимое этого файла в базе данных, с тем чтобы модуль GeoDjango мог с ним работать. Поскольку заранее мы не знаем, какие типы геометрий будут содержаться в файле фигур или какие атрибуты могут быть связаны с каждым геообъектом, база данных должна располагать универсальным представлением содержимого файла фигур вместо задания отдельных полей для каждого атрибута файла фигур.

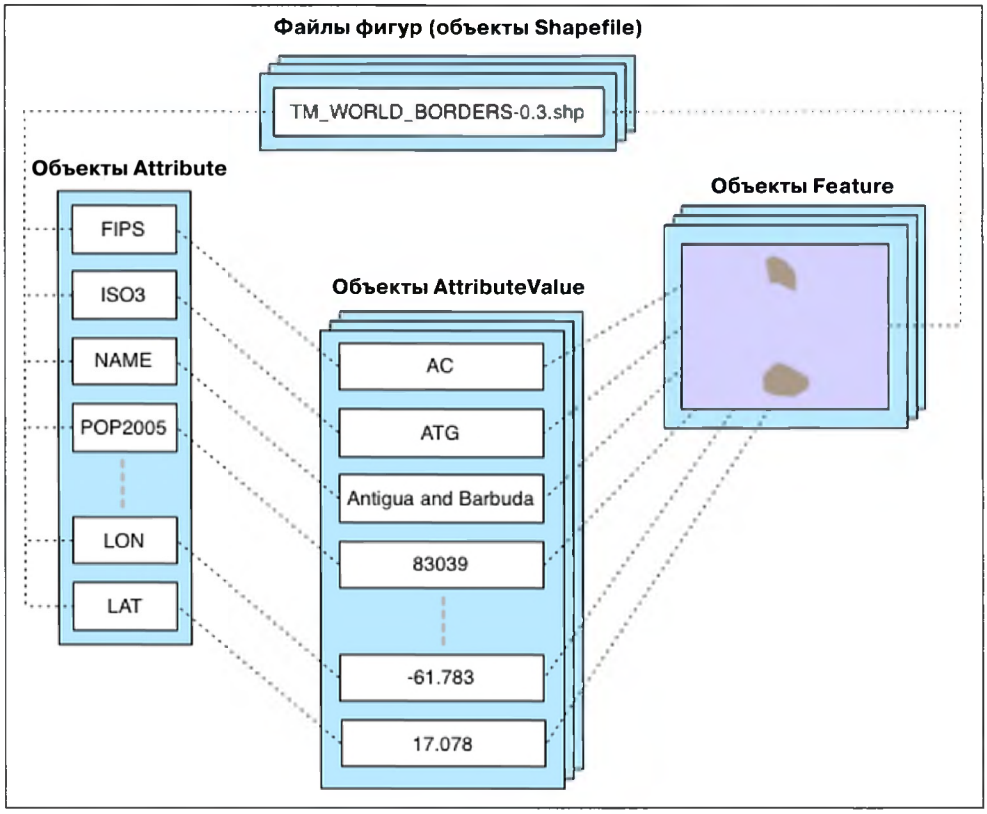
Чтобы поддержать эту функциональность, мы воспользуемся следующим набором объектов базы данных:



Каждый импортированный файл фигур в базе данных будет представлен одним объектом `Shapefile`. Объект `Shapefile` будет иметь набор объектов `Attribute`, которые задают имя и тип данных для каждого атрибута файла фигур. Объект

Shapefile также будет иметь набор объектов Feature (Геообъект), один для каждого импортированного геообъекта. Объекты Feature будут содержать геометрии геообъектов файла фигур, при этом объект Feature будет также иметь набор объектов AttributeValue (Значение атрибута), которые будут содержать значения атрибутов геообъекта.

Чтобы понять, как это работает, предположим, что мы импортируем в систему ShapeEditor набор данных границ стран мира. Тогда содержимое соответствующего файла фигур будет сохранено в базе данных следующим образом:



Мы будем работать с объектом Shapefile, представляющим загруженный файл фигур. Этот объект имеет ряд связанных с ним объектов Attribute, один для каждого атрибута файла фигур. Кроме того, также имеется ряд связанных с ним объектов Feature: геометрия составного многоугольника MultiPolygon каждого геообъекта будет храниться непосредственно в самом объекте Feature, в то время как атрибуты геообъекта будут храниться в серии объектов AttributeValue.

Несмотря на то что это несколько окольный способ хранения данных файла фигур в базе данных (было бы легче применить команду управления ogrinspect, чтобы

создать из геообъектов и атрибутов файла фигур статическую модель GeoDjango), мы вынуждены поступить именно так, потому что мы заранее не знаем структуру файла фигур, а определять новую таблицу базы данных всякий раз, когда файл фигур импортируется, совершенно непрактично.

При такой исходной модели представления данных файла фигур в базе данных мы можем продолжить разработку остальной части логики *импортирования файла фигур*.

Поскольку файлы фигур на диске представлены несколькими отдельными файлами, мы ожидаем, что пользователь создаст из файла фигур архив ZIP и выгрузит его на сервер в заархивированном виде. Это избавляет нас от необходимости обрабатывать выгрузку нескольких файлов, относящихся к одному файлу фигур, и сделает все удобнее для пользователя, поскольку файлы фигур часто уже идут в формате ZIP.

После того как архив ZIP выгружен на сервер, наш программный код должен распаковать архив и извлечь оттуда отдельные файлы, которые составляют файл фигур. Затем нам нужно прочитать файл фигур, чтобы найти его атрибуты, создать соответствующие объекты `Attribute` и затем поочередно обработать геообъекты файла фигур, создавая по мере продвижения объекты `Feature` и `AttributeValue`. Все это реализуется достаточно прямолинейно.

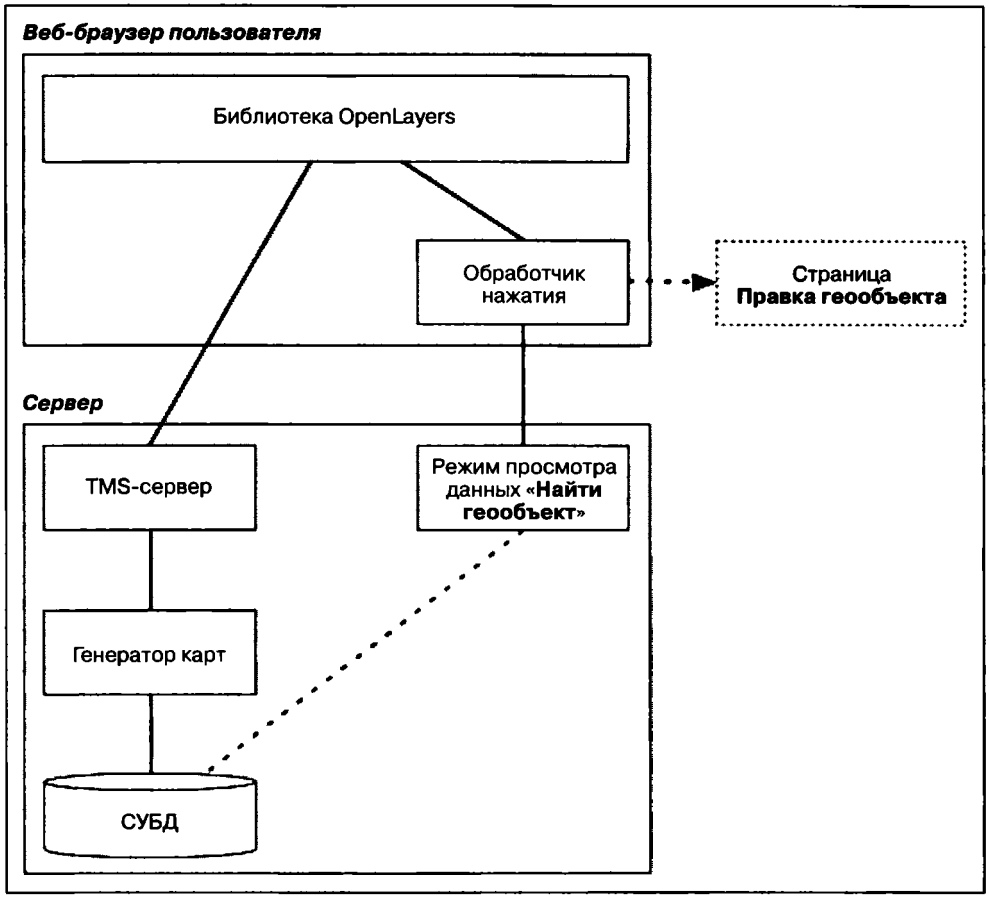
Выбор геообъекта

Прежде чем пользователь сможет отредактировать геообъект, мы должны обеспечить пользователю возможность его выбрать. К сожалению, встроенный в модуль GeoDjango интерфейс скользящей карты не позволяет нам выбрать геообъект, нажав на нем мышью. Это объясняется особенностью организации в нем поля геометрии, из-за которого он способен за один раз выводить на карте всего один геообъект.

Традиционно приложение модуля GeoDjango позволяет выбрать геообъект, выводя на экран список атрибутов (например, названий городов) и затем позволяя выбрать геообъект из этого списка. К сожалению, в нашем случае это тоже не будет работать. Поскольку система ShapeEditor дает возможность пользователю импортировать *любой* файл фигур, нет никакой гарантии, что при помощи значений его атрибутов получится выбрать геообъект. Может случиться так, что файл фигур не имеет никаких атрибутов вообще либо имеет атрибуты, которые для конечного пользователя ничего не означают, или, наоборот имеет десятки атрибутов. Невозможно предсказать, какой атрибут надо показывать, или даже есть ли вообще подходящий атрибут, при помощи которого можно выбрать геообъект. Именно по этой причине при выборе геообъекта для правки вывести список атрибутов не получится.

Вместо этого мы проявим абсолютно другой подход. Мы обойдем встроенный в GeoDjango редактор и вместо него воспользуемся библиотекой OpenLayers, чтобы вывести карту напрямую, показывая абсолютно *все* геообъекты импортированного файла фигур. Затем мы дадим пользователю возможность щелкнуть по геообъекту внутри карты, чтобы выбрать его для редактирования.

Ниже показано то, как мы собираемся реализовать этот конкретный функционал:



Библиотеке OpenLayers нужно иметь источник сборных карт для отображения на экране компьютера, поэтому мы создадим наш собственный простой сервер сборных цифровых карт для показа хранящихся в базе данных геообъектов файла фигур, надстроенный поверх генератора карт и реализованный на основе библиотеки Mapnik. Мы также напишем простой **обработчик события** click на JavaScript, который перехватывает нажатия на карте и отсылает запрос AJAX на сервер, чтобы установить, на каком объекте пользователь щелкнул мышью. И если пользователь действительно щелкнул на геообъекте (а не на подложке карты), то веб-браузер пользователя будет перенаправлен на страницу **правки геообъекта**, с тем чтобы он мог отредактировать геообъект, находящийся в месте нажатия.

Здесь требуется изрядное количество обычного программирования, но в конечном итоге получится дружественный интерфейс системы ShapeEditor, позволяющий пользователю отредактировать геообъект, всего-навсего наведя на него

курсор и щелкнув по нему мышью. В процессе создания всего этого мы также научимся использовать библиотеку OpenLayers непосредственно изнутри пользовательского приложения модуля GeoDjango и реализовывать свой собственный сервер сборных цифровых карт, создаваемый поверх библиотеки Mapnik.

Правка геообъекта

Чтобы позволить пользователю отредактировать геообъект, мы будем использовать встроенный в GeoDjango виджет редактирования геометрий. Здесь потребуется небольшой объем работы, потому что мы хотим использовать этот виджет за пределами веб-интерфейса администратора GeoDjango, и поэтому нам придется немного поработать над его настройками.

Еще одна и последняя проблема, которую необходимо решить, заключается в том, что мы не знаем заранее, какой геообъект мы будем редактировать. Файлы фигур могут содержать любой тип геометрии от точек и ломаных до составных многоугольников и коллекций геометрий. К счастью, все геообъекты в файле фигур имеют одинаковый тип геометрии, поэтому мы можем сохранить тип геометрии в объекте `Shapefile` и использовать его для выбора соответствующего типа редактора во время правки геообъектов файла фигур.

Экспорт файла фигур

Экспорт файла фигур подразумевает операцию, обратную процессу импорта: мы должны создать на диске новый файл фигур, задать различные атрибуты, которые будут сохранены в файле фигур, и затем обработать все геообъекты и их атрибуты, записав их в файл фигур. После того как это будет сделано, мы можем упаковать содержимое файла фигур в архив ZIP и сообщить веб-браузеру пользователя скачать этот архив ZIP на жесткий диск пользователя.

Необходимые компоненты

Прежде чем мы сможем начать реализовывать систему ShapeEditor, мы должны убедиться, что на вашем компьютере установлены различные библиотеки и инструменты, которые она использует. Скорее всего, вы их все уже установили, когда работали с предыдущими главами этой книги, но на случай, если вы этого не сделали, пожалуйста, удостоверьтесь, что вы установили следующее программное обеспечение:

- программный пакет GDAL/OGR версии 1.10 или выше;
- динамическая библиотека PROJ.4 версии 4.8 или выше;
- библиотека Python pyproj версии 1.9 или выше;
- СУБД PostgreSQL версии 9.4 или выше;
- геопространственное расширение PostGIS версии 2.1 или выше;
- адаптер СУБД PostgreSQL для Python psycopg2 версии 2.6 или выше;
- динамическая библиотека Mapnik версии 2.2 или выше;
- веб-платформа Django версии 1.8 или выше.

Настройка базы данных

Поскольку система ShapeEditor интенсивно использует расширение PostGIS, сначала нам нужно выполнить настройку пользователя СУБД PostgreSQL и базы данных для использования в веб-приложении ShapeEditor, а затем активировать расширение PostGIS для этой базы данных. Сейчас мы этим и займемся.

1. Откройте окно терминала или командной строки и наберите следующую команду:

```
createuser -P shapeeditor
```

Напомним, что следует добавить параметр командной строки `-U postgres` либо воспользоваться утилитой `sudo` для этой команды, если необходимо запустить СУБД Postgres под другой учетной записью пользователя.

2. Вам будет предложено ввести пароль для пользователя СУБД Postgres `shapeeditor`. Убедитесь в правильности вводимого вами пароля, так как он вам потребуется, когда будете настраивать веб-приложение ShapeEditor, с тем чтобы оно могло обращаться к базе данных. Далее нам нужно создать непосредственно саму базу данных:

```
% createdb shapeeditor
```

Опять же, добавьте параметр командной строки `-U` либо, если нужно, воспользуйтесь утилитой `sudo`.

3. Затем нужно сообщить СУБД Postgres, что пользователь `shapeeditor` может получать доступ к базе данных `shapeeditor`:

```
% psql -c "GRANT ALL PRIVILEGES ON DATABASE shapeeditor TO
shapeeditor; "
```

4. Наконец, нужно пространственно активировать базу данных `shapeeditor`, подключив расширение PostGIS:

```
% psql -d shapeeditor -c "CREATE EXTENSION postgis;"
```

Для всех этих команд добавьте параметр командной строки `-U` либо, если нужно, воспользуйтесь утилитой `sudo`.

Поздравляем! Вы только что настроили базу данных PostGIS для использования в веб-приложении ShapeEditor.

Настройка проекта ShapeEditor

Теперь нам нужно создать для нашей системы ShapeEditor проект Django. Для этого при помощи команды `cd` перейдите в каталог, куда вы хотите разместить проект, и наберите следующее:

```
% django-admin.py startproject shapeEditor
```

Когда вы инсталлировали Django, он должен был поместить программу `django-admin.py` в ваш системный путь, поэтому вам не нужно сообщать компьютеру, где этот сценарий находится.

Если все выполнено правильно, Django создаст каталог `shapeEditor` со следующим содержимым:

```
manage.py
shapeEditor/
    __init__.py
    settings.py
    urls.py
    wsgi.py
```

Теперь, когда проект создан, мы должны его сконфигурировать. Для этого отредактируйте файл `settings.py`, который находится в каталоге пакета shapeEditor. Мы должны сообщить нашему проекту, как получать доступ к базе данных, которую мы настроили ранее, а также активировать модуль расширения GeoDjango.

Начнем с поиска переменной `DATABASES` и изменим ее так, чтобы она выглядела, как показано ниже:

```
DATABASES = {
    'default': {
      'ENGINE' : 'django.contrib.gis.db.backends.postgis',
      'NAME' : 'shapeeditor',
      'USER' : 'shapeeditor',
      'PASSWORD' : '...'
    }
}
```

Удостоверьтесь, что вы вводите пароль, который вы настроили для пользователя СУБД Postgres в `shapeeditor`.

Затем отыщите переменную `INSTALLED_APPS` и добавьте в конец списка установленных приложений следующую ниже строку:

```
'django.contrib.gis',
```

Пока мы редактируем файл `settings.py`, внесем еще одно изменение, которое избавит нас от некоторых проблем в дальнейшем. Перейдите в настройки `MIDDLEWARE_CLASSES` и закомментируйте строку `django.middleware.csrf.CsrfViewMiddleware`. Эта запись вызывает добавление лишней проверки на ошибки во время обработки формы, чтобы предотвратить подделку кросс-сайтовых запросов, англ. термин cross-site request forgery (CSRF). Реализация поддержки CSRF, которую мы не будем тут реализовывать, чтобы не усложнять разработку, требует добавления в наши шаблоны форм лишнего программного кода.

Если вы развертываете свои собственные приложения в Интернете, то следует почитать документацию по CSRF на веб-сайте Django и включить поддержку CSRF. Иначе ваше приложение может подвергнуться атакам с использованием подделки кросс-сайтовых запросов.

На этом конфигурирование нашего проекта ShapeEditor завершено.

Определение приложений ShapeEditor

Теперь в нашем распоряжении имеется проект Django для всей нашей системы ShapeEditor. Далее нам нужно подразделить наш проект на приложения, согласно принципам проектирования веб-платформы Django, которые требуют, чтобы приложения были небольшими и относительно автономными. Анализируя дизайн всего нашего проекта, мы видим нескольких возможных кандидатов для разбиения функциональности на отдельные приложения:

- импорт и экспорт файлов фигур;
- выбор геообъектов;
- редактор геообъектов;
- сервер сборных цифровых карт.

Мы объединим первые три в отдельное приложение под названием shapefiles, которое возьмет на себя всю логику, связанную с файлом фигур. Затем у нас будет еще одно приложение – tms, в котором будет реализован наш сервер сборных цифровых карт. Наконец, мы определим еще одно приложение, которое мы назовем shared, в котором будут содержаться общие для этих приложений модели базы данных и модули Python.

Например, у нас может иметься модуль utils.py, который нужен как для приложения shapefiles, так и для приложения tms. Мы поместим его в общее приложение shared, чтобы ясно дать понять, что этот программный код разработан для совместного использования различными частями системы.

Создание общего приложения shared

Приложение shapeEditor.shared будет содержать основные таблицы базы данных и модули Python, которые мы используем по всей системе. Пойдем дальше и теперь создадим это приложение. При помощи команды cd поменяйте текущий каталог на верхний каталог shapeEditor и наберите следующую команду:

```
python3 manage.py startapp shared
```

В результате будет создан новый пакет Python с именем shared, в котором будет находиться содержимое приложения shared. Отметим, что по умолчанию новое приложение помещается в верхний каталог приложения shapeEditor. Это означает, что это приложение можно импортировать в программу на Python, как показано ниже:

```
import shared
```

Согласно соглашениям Django об условиях использования приложений, предполагается, что приложения в верхнем каталоге (или где-либо еще в системном пути вашей среды программирования Python) являются приложениями *многократного использования* – то есть можно взять это приложение и использовать в другом проекте.

Приложения, которые мы определяем здесь, не являются таковыми; они могут работать только в составе проекта `shapeEditor`, и мы хотели бы импортировать их, как показано ниже:

```
import shapeEditor.shared
```

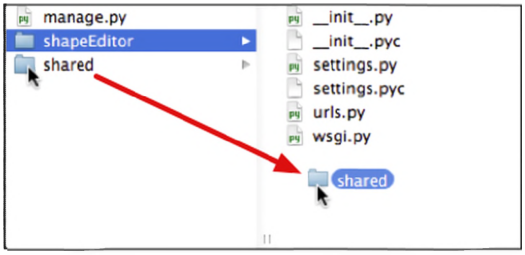
Чтобы это стало возможным, нужно переместить недавно созданный каталог пакета `shared` в подкаталог проекта `shapeEditor`. Это легко сделать из командной строки; для этого при помощи команды `cd` просто перейдите во внешний каталог `shapeEditor` и наберите следующую команду:

```
mv shared shapeEditor
```

Как вариант, если вы работаете под Windows, вам следует воспользоваться нижеследующей командой:

```
move shared shapeEditor
```

Или просто перетащите папку, как показано ниже:



К сожалению, веб-платформа Django в настоящее время несколько затрудняет создание приложения немногократного использования. Чтобы приложение не выполнялось многократно, его сначала нужно создать и затем переместить его каталог в каталог проекта.

В каталоге пакета `shapeEditor.shared` будет ряд файлов:

Файл	Описание
`init.py`	Стандартный файл инициализации пакета Python
`admin.py`	Модуль, который задает веб-интерфейс администратора для приложения shapeEditor.shared
`migrations`	Каталог, в котором будут содержаться различные миграции базы данных, используемые в этом приложении
`models.py`	Модуль Python, который содержит модели базы данных приложения shapeEditor. shared
`tests.py`	Модуль, который может содержать модульные тесты приложения
`views.py`	Модуль, который обычно содержит просмотровые функции приложения

Пойдем дальше и удалим модули `tests.py` и `views.py`, поскольку они для этого приложения не понадобятся. В итоге вы должны получить нижеследующую структуру каталогов:

```
manage.py
shapeEditor/
    __init__.py
    settings.py
    shared/
        __init__.py
        admin.py
        migrations/
            __init__.py
        models.py
    urls.py
    wsgi.py
```

Поскольку мы создали непосредственно само приложение, давайте добавим его к нашему проекту. Снова откройте файл `settings.py` и добавьте в список `INSTALLED_APPS` следующую ниже запись:

```
'shapeEditor.shared',
```

Имея в нашем распоряжении общее приложение `shapeEditor.shared`, теперь начнем придавать ему нужную форму.

Определение моделей данных

Мы уже знаем, какие объекты базы данных нам потребуются для хранения выгруженных на сервер файлов фигур:

- объект `Shapefile` будет представлять один файл фигур, выгруженный на сервер с клиентского компьютера;
- каждый объект `Shapefile` будет иметь ряд объектов `Attribute`, сообщая об имени, типе данных и другой информации обо всех атрибутах внутри файла фигур;
- каждый объект `Shapefile` будет иметь ряд объектов `Feature`, которые содержат геометрию всех геообъектов файла фигур;
- каждый объект `Feature` будет иметь ряд объектов `AttributeValue`, которые содержат значения всех атрибутов геообъектов.

Рассмотрим каждый из них подробнее и подумаем о том, какая именно информация должна храниться в каждом объекте.

Объект Shapefile

При импортировании файла фигур следует запомнить следующую ниже информацию:

- исходное имя выгруженного файла. Мы покажем его в режиме просмотра данных «Вывести список файлов фигур», с тем чтобы пользователь мог выбрать файл фигур из этого списка;
- используемая в файле фигур система пространственной привязки. Когда мы будем импортировать файл фигур, мы выполним его конвертацию,

с тем чтобы воспользоваться географическими координатами широты и долготы с использованием датума WGS84 (EPSG 4326), но при этом мы должны помнить об исходной системе пространственной привязки файла фигур, чтобы можно было снова ей воспользоваться при экспорте геообъектов. Для простоты мы будем хранить систему пространственной привязки в формате WKT;

- тип геометрии, хранящейся в файле фигур. Нам это понадобится, чтобы знать, какое поле в объекте `Feature` содержит геометрию.

Объект Attribute

При экспорте файла фигур он должен иметь те же самые атрибуты, что и исходный импортированный файл. По этой причине мы должны помнить атрибуты файла фигур.

Объект `Attribute` именно это и делает. По каждому атрибуту нам нужно запомнить нижеследующую информацию:

- какому файлу фигур атрибут принадлежит;
- имя атрибута;
- хранящийся в этом атрибуте тип данных (строковое значение, число с плавающей точкой и т. д.);
- ширина поля атрибута в символах;
- число знаков после десятичной точки для отображения (для атрибутов с плавающей точкой).

Вся эта информация поступает непосредственно из определения слоя файла фигур.

Объект Feature

Каждый геообъект в импортированном файле фигур должен быть сохранен в базе данных. Поскольку в расширении PostGIS (и модуле GeoDjango) для разных типов геометрий используются разные типы полей, для каждого типа геометрии мы должны задать отдельные поля. Вследствие этого объект `Feature` должен хранить следующую информацию:

- файл фигур, которому геообъект принадлежит;
- поле для геометрии точки, если в файле фигур хранится этот тип геометрии;
- поле для геометрии составной точки, если в файле фигур хранится этот тип геометрии;
- поле для геометрии составной ломаной, если в файле фигур хранится этот тип геометрии;
- поле для геометрии составного многоугольника, если в файле фигур хранится этот тип геометрии;
- поле для геометрии с коллекцией геометрий, если в файле фигур хранится этот тип геометрии.

Что-то пропущено?

Вы, наверное, обратили внимание, что нескольких типов геометрий не хватает. А как быть с многоугольниками и ломаными? В силу того, как организовано хранение данных в файле фигур, невозможно знать заранее, содержатся ли там многоугольники или составные многоугольники, как и то, содержатся ли там ломаные или составные ломаные (а также точки или составные точки). Внутренняя структура файла фигур не делает различия между этими типами геометрий. По этой причине можно ошибочно принять, что в нем хранятся многоугольники, тогда как в действительности там находятся составные многоугольники; то же самое можно сказать и в отношении геометрий ломаных линий. Для получения дополнительной информации обратитесь к http:// code.djangoproject.com/ticket/7218.

Чтобы обойти это ограничение, мы храним все многоугольники, ломаные и точки как составные многоугольники, ломаные и точки. Именно по этой причине в объекте `Feature` нам не нужны поля для многоугольника, ломаной и точки.

Объект AttributeValue

Объект `AttributeValue` содержит значение каждого атрибута геообъекта. Этот объект довольно прост и хранит следующую информацию:

- какому геообъекту принадлежит значение атрибута;
- какому атрибуту принадлежит значение;
- значение атрибута в виде строки символов в кодировке Юникода.

Для простоты мы будем хранить все значения атрибута как строковые.

Файл models.py

Зная, какую информацию мы хотим хранить в нашей базе данных, теперь легко определить наши объекты модели. Для этого отредактируйте файл `models.py`, находящийся в каталоге `shapeEditor.shared`, и удостоверьтесь, что он выглядит, как показано ниже:

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

class Shapefile(models.Model):
    filename  = models.CharField(max_length=255)
    srs_wkt   = models.CharField(max_length=255)
    geom_type = models.CharField(max_length=50)

class Attribute(models.Model):
    shapefile = models.ForeignKey(Shapefile)
    name      = models.CharField(max_length=255)
    type      = models.IntegerField()
    width     = models.IntegerField()
    precision = models.IntegerField()

class Feature(models.Model):
    shapefile  = models.ForeignKey(Shapefile)
```

```
    geom_point = models.PointField(srid=4326,
                                   blank=True, null=True)
    geom_multipoint = \
      models.MultiPointField(srid=4326,
                             blank=True, null=True)
    geom_multilinestring = \
      models.MultiLineStringField(srid=4326,
                                  blank=True, null=True)
    geom_multipolygon = \
      models.MultiPolygonField(srid=4326,
                               blank=True, null=True)
    geom_geometrycollection = \
      models.GeometryCollectionField(srid=4326,
                                     blank=True,
                                     null=True)
    objects = models.GeoManager()

class AttributeValue(models.Model):
    feature   = models.ForeignKey(Feature)
    attribute = models.ForeignKey(Attribute)
    value     = models.CharField(max_length=255,
                                 blank=True, null=True)
```

Здесь необходимо знать следующее.

- Оператор импорта `from...import` вверху изменился. Вместо стандартных моделей Django мы импортируем модели GeoDjango.
- Мы используем объекты `models.CharField` для представления символьных данных и `models.IntegerField` для представления целочисленных значений. Django предоставляет пользователю целый ряд типов полей. Помимо них, модуль GeoDjango добавляет свои собственные типы полей, которые предназначены для хранения полей геометрии, как видно из определения объекта `Feature`.
- Для представления связей между двумя объектами мы используем объект `models.ForeignKey`.
- Учитывая, что в объекте `Feature` будут храниться данные геометрий, мы хотим, используя эти данные, позволить GeoDjango выполнять пространственные запросы. Чтобы активировать эту функциональность, мы определяем для класса `Feature` экземпляр `GeoManager()`.
- Несколько полей (в частности, поля `geom_XXX` в объекте `Feature`) имеет элементы `blank=True` и `null=True`. В действительности их смысл сильно различается: `blank=True` означает, что веб-интерфейс администратора разрешает пользователю оставлять поле пустым, в то время как `null=True` сообщает базе данных, что эти поля в ней можно установить в `NULL`. Для объекта `Feature` нам потребуются оба поля, с тем чтобы мы не получали ошибок валидации при вводе геометрий через веб-интерфейс администратора.

И это все, что (на данный момент) нам нужно сделать, чтобы сформулировать наши модели базы данных. После того как вы внесли эти изменения, сохраните

файл, перейдите при помощи команды cd в верхний каталог проекта и наберите следующее:

```
python3 manage.py makemigrations shared
```

Как мы убедились в предыдущей главе, команда makemigrations создает для заданного приложения файл миграции базы данных. При наличии этой миграции мы теперь можем поручить веб-платформе Django создать структуру базы данных, которая будет соответствовать содержимому наших моделей базы данных:

```
python3 manage.py migrate
```

Эта команда поручает веб-платформе Django проверить модели и создать новые таблицы базы данных, как того требуется. Теперь у вас должна быть пространственная база данных, в которой различные таблицы созданы согласно вашим определениям. Приглядимся к этой базе данных поближе, набрав следующую ниже команду:

```
psql shapeeditor
```

Когда вы увидите командную строку СУБД Postgres, наберите \d и нажмите *Return*. Вы должны увидеть список всех созданных таблиц базы данных:

```
                    Список отношений
Схема  | Имя                                | Тип                | Владелец
-------+------------------------------------+--------------------+---------
public | auth_group                         | таблица            | shapeeditor
public | auth_group_id_seq                  | последовательность | shapeeditor
public | auth_group_permissions             | таблица            | shapeeditor
public | auth_group_permissions_id_seq      | последовательность | shapeeditor
public | auth_permission                    | таблица            | shapeeditor
public | auth_permission_id_seq             | последовательность | shapeeditor
public | auth_user                          | таблица            | shapeeditor
public | auth_user_groups                   | таблица            | shapeeditor
public | auth_user_groups_id_seq            | последовательность | shapeeditor
public | auth_user_id_seq                   | последовательность | shapeeditor
public | auth_user_user_permissions         | таблица            | shapeeditor
public | auth_user_user_permissions_id_seq  | последовательность | shapeeditor
public | django_admin_log                   | таблица            | shapeeditor
public | django_admin_log_id_seq            | последовательность | shapeeditor
public | django_content_type                | таблица            | shapeeditor
public | django_content_type_id_seq         | последовательность | shapeeditor
public | django_migrations                  | таблица            | shapeeditor
public | django_migrations_id_seq           | последовательность | shapeeditor
public | django_session                     | таблица            | shapeeditor
public | geography_columns                  | представление      | postgres
public | geometry_columns                   | представление      | postgres
public | raster_columns                     | представление      | postgres
public | raster_overviews                   | представление      | postgres
public | shared_attribute                   | таблица            | shapeeditor
```

```
public | shared_attribute_id_seq          | последовательность | shapeeditor
public | shared_attributevalue            | таблица            | shapeeditor
public | shared_attributevalue_id_seq     | последовательность | shapeeditor
public | shared_feature                   | таблица            | shapeeditor
public | shared_feature_id_seq            | последовательность | shapeeditor
public | shared_shapefile                 | таблица            | shapeeditor
public | shared_shapefile_id_seq          | последовательность | shapeeditor
public | spatial_ref_sys                  | таблица            | postgres
(32 строки)
```

Чтобы обеспечить в базе данных уникальность таблиц каждого приложения, веб-платформа Django добавляет имя приложения к началу имени таблицы. Это означает, что имена таблиц для моделей, которые мы создали внутри общего приложения `shared`, будут на самом деле называться `shared_shapefile`, `shared_feature` и т. д. Мы будем работать непосредственно с этими таблицами базы данных позже, когда нам понадобится библиотека Mapnik для визуализации цифровой карты на основе импортированных данных файла фигур.

Когда вы закончите работу с клиентской утилитой командной строки СУБД Postgres, наберите `\q` и нажмите *Return*, чтобы выйти из `psql`.

Настроив нашу базу данных, теперь создадим учетную запись «суперпользователя», чтобы мы могли получать к ней доступ. Это можно сделать, набрав ниже следующую команду:

```
python3 manage.py createsuperuser
```

Вам будет предложено ввести имя пользователя, адрес электронной почты и пароль для нового суперпользователя; он нам пригодится для следующего раздела, где мы займемся изучением встроенного веб-интерфейса администратора GeoDjango.

Знакомство с подсистемой администрирования

Встроенное приложение администрирования **admin** включается в новые проекты веб-платформы Django по умолчанию. Тем не менее, прежде чем мы сможем им воспользоваться, мы должны зарегистрировать различные модели базы данных, которые мы хотим, чтобы оно поддерживало. Для этого откройте в текстовом редакторе модуль `admin.py` внутри каталога `shapeEditor/shared` и наберите в этот файл следующий ниже фрагмент программного кода:

```
from django.contrib.gis import admin
from shapeEditor.shared.models import *

admin.site.register(Shapefile, admin.ModelAdmin)
admin.site.register(Feature, admin.GeoModelAdmin)
admin.site.register(Attribute, admin.ModelAdmin)
admin.site.register(AttributeValue, admin.ModelAdmin)
```

Класс `ModelAdmin` сообщает веб-платформе Django, как показывать модель внутри веб-интерфейса администратора. Отметим, что для класса `Feature` мы используем класс `GeoModelAdmin`. Учитывая, что объект `Feature` содержит поля геометрий, использование класса `GeoModelAdmin` позволяет, находясь в веб-интерфейсе администратора, редактировать эти поля геометрии при помощи скользящей карты. Вскоре мы увидим, как это работает.

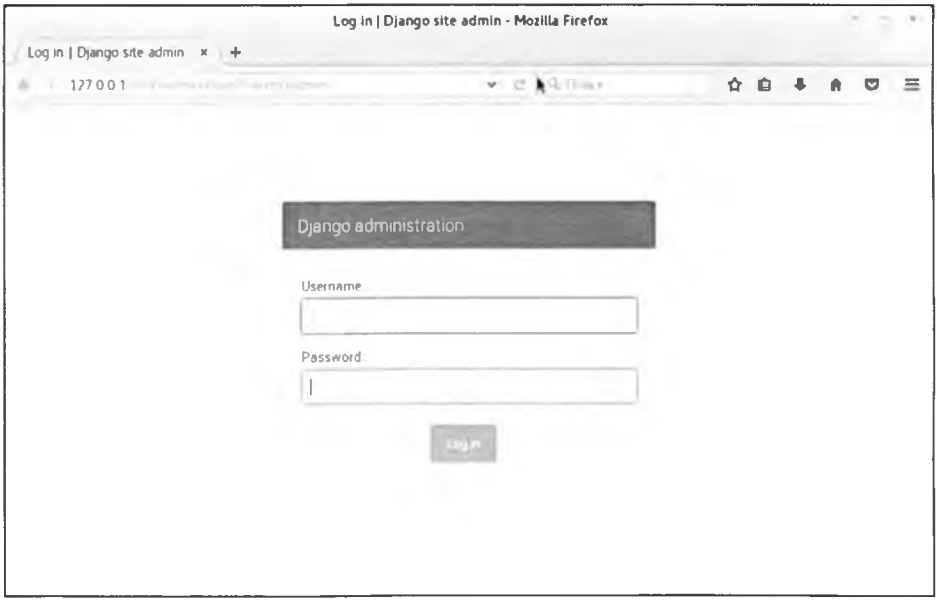
Сконфигурировав административный модуль, теперь попытаемся его выполнить. Наберите в своем окне терминала или командной строки следующую ниже команду:

```
python3 manage.py runserver
```

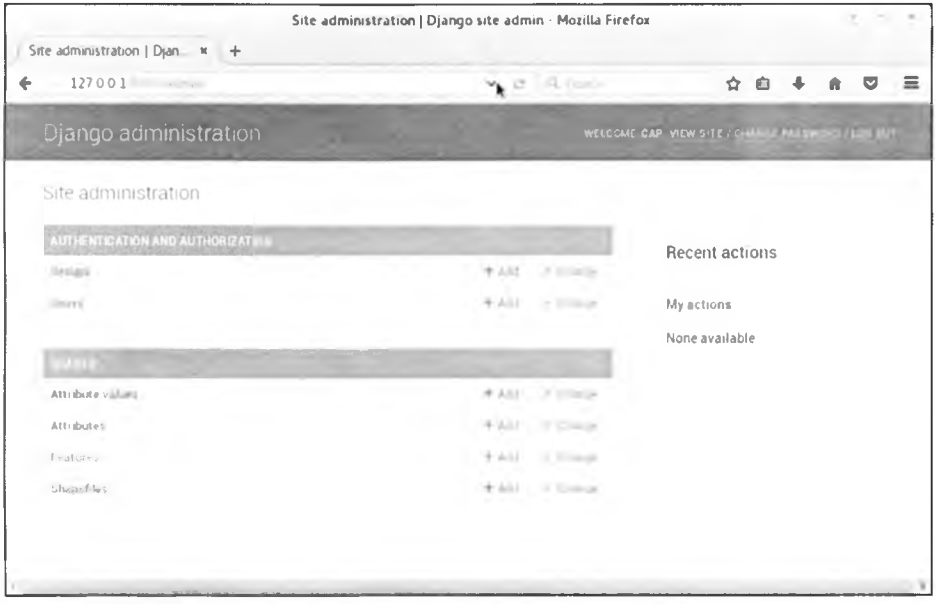
В результате для вашего проекта будет запущен сервер Django. Откройте веб-браузер и перейдите по следующему URL-адресу:

```
http://127.0.0.1:8000/admin/shared
```

Вы должны увидеть страницу авторизации в **административном разделе** приложения веб-платформы Django:

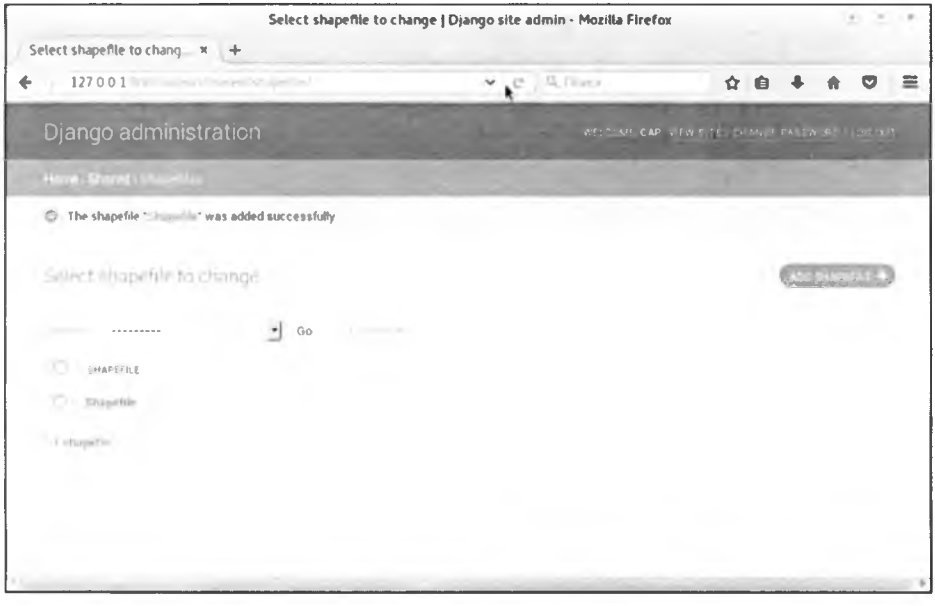


Введите имя пользователя и созданный вами ранее пароль суперпользователя, и вы увидите главную страницу веб-интерфейса администратора общего приложения `shapeEditor.shared`:



Воспользуемся этим веб-интерфейсом администратора, чтобы создать фиктивный файл фигур. Нажмите на ссылку **Add** (Добавить) в строке **Shapefiles** (Файлы фигур), и вам будет предложен основной экран ввода нового файла фигур:

Введите в разные поля несколько фиктивных значений (не имеет значения, что вы вводите) и щелкните по кнопке **Save** (Сохранить), чтобы сохранить новый объект Shapefile в базе данных. Будет показан список присутствующих в базе данных файлов фигур. В данный момент имеется только одна запись Shapefile, которую вы только что создали:



Как видите, для обозначения нового объекта файла фигур показана довольно бесполезная надпись: **объект Shapefile**. Это объясняется тем, что мы еще не указали веб-платформе Django, какую надпись использовать для файла фигур. Чтобы ее скорректировать, откройте в текстовом редакторе файл `shared.models` и добавьте в конец определения класса Shapefile следующий ниже метод:

```
def __str__(self):
    return self.filename
```

Метод `__str__` возвращает удобочитаемое резюме о содержании объекта Shapefile. В этом случае мы показываем связанное с файлом фигур имя файла. Если затем перезагрузить веб-страницу, то вы увидите, что файл фигур теперь имеет приемлемую надпись:

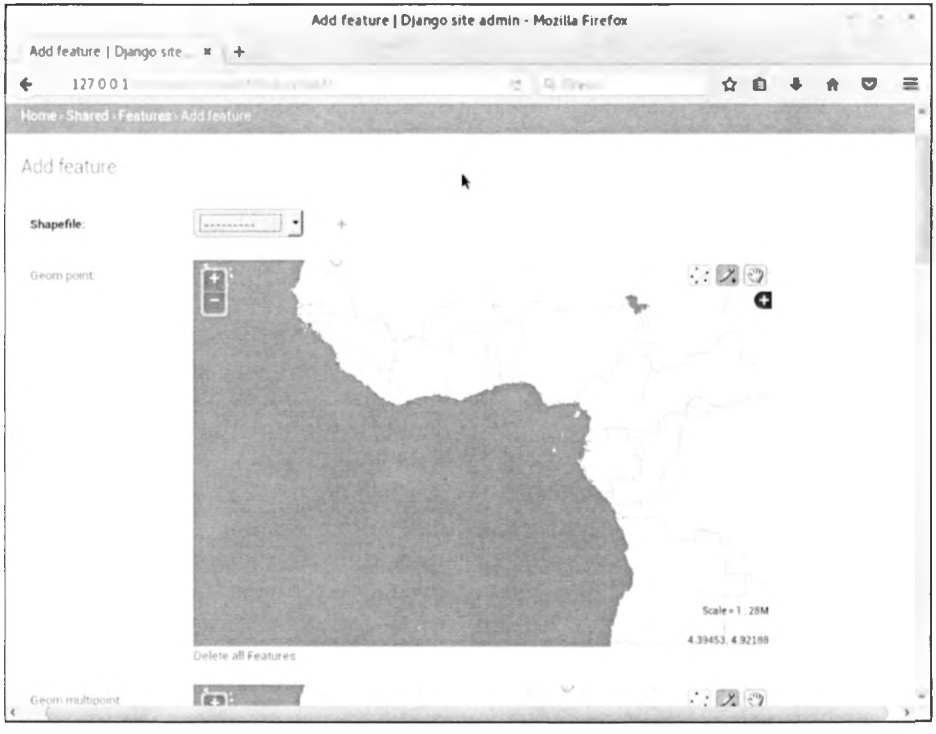
Пойдем дальше и добавим метод `__str__` к другим объектам модели:

```
class Attribute(models.Model):
    ...
    def __str__(self):
        return self.name

class Feature(models.Model):
    ...
    def __str__(self):
        return str(self.id)

class AttributeValue(models.Model):
    ...
    def __str__(self):
        return self.value
```

На первый взгляд это может показаться бесполезной работой, вместе с тем вашим объектам базы данных на самом деле совсем не повредит функционал самоописания. Если потребуется, вы можете расширить настройку веб-интерфейса администратора, например добавив отображение атрибутов и геообъектов, связанных с выбранным файлом фигур. А пока, впрочем, взглянем на встроенные в GeoDjango редакторы геометрий.

Вернитесь к странице администрирования общего приложения `shared` (нажав на гиперссылку **Shared** в верхней части окна) и щелкните по кнопке **Add** (Добавить) в строке **Features** (Геообъекты). Как и в случае с файлом фигур, вам будет предложено ввести подробные данные о новом геообъекте. Правда, на этот

раз в веб-интерфейсе администратора для ввода всех поддерживаемых объектом `Feature` типов геометрий будет использована скользящая карта:



Очевидно, наличие нескольких скользящих карт, как показано на снимке экрана, – это не вполне то, что мы хотим, и, если бы было нужно, мы бы могли настроить специальный подкласс `GeoModelAdmin` так, чтобы этого избежать, но прямо сейчас это не принципиально. Вместо этого попробуйте выбрать файл фигур, чтобы связать с этим геообъектом, выбрав созданный файл фигур из меню **Shapefile**, и затем прокрутите вниз к полю составного многоугольника **Geom multipolygon** и попробуйте добавить в карту один или два многоугольника. Для этого щелкните по карте, чтобы начать отрисовку нового многоугольника, щелкните несколько раз, чтобы добавить точки к текущему многоугольнику, либо сделайте щелчок, удерживая клавишу *Shift*, чтобы закончить создание многоугольника. Поначалу веб-интерфейс может показаться немного запутанным, но он, конечно же, пригоден для использования. Позже мы обратимся к различным вариантам редактирования многоугольников. На данный момент просто щелкните по кнопке **Save** (Сохранить) в самом конце страницы, чтобы сохранить ваш новый геообъект. Если вы будете редактировать его снова, то на скользящих картах увидите свою сохраненную геометрию (или геометрии) еще раз.

На этом мы завершим наш тур по веб-интерфейсу администратора. Мы не будем использовать его для конечных пользователей, поскольку перед нами не стоит задача требовать от пользователей авторизации перед внесением изменений в данные файла фигур. Однако у приложения администрирования мы все же позаимствуем немного программного кода, с тем чтобы конечные пользователи могли отредактировать геообъекты своего файла фигур при помощи скользящей карты.

Заключение

Вы только что закончили реализацию первой части приложения ShapeEditor. Даже на этой ранней стадии вы добились достаточного прогресса, узнав, как работает модуль GeoDjango, разработав приложение и заложив основы функциональности, которую вы воплотите в следующих двух главах.

В этой главе вы создали свой собственный проект GeoDjango, подробно разработали систему ShapeEditor и разложили его на отдельные приложения в рамках проекта веб-платформы Django. Вы определили ряд моделей базы данных, которые будут использоваться в системе ShapeEditor, и настроили базу данных PostGIS, в которой будут храниться данные этой системы. Затем вы сконфигурировали встроенное приложение администрирования, чтобы оно могло просматривать и редактировать задаваемые вами модели базы данных, и вы применили это приложение для просмотра и правки ваших данных. Наконец, вы увидели, как класс `GeoModelAdmin` модуля GeoDjango позволяет пользователю просматривать и редактировать геоданные, используя для этого скользящую карту.

В следующей главе мы займемся реализацией режима просмотра данных для вывода списка имеющихся файлов фигур, а также напишем программный код более высокой сложности для импорта и экспорта файлов фигур через веб-интерфейс.

Глава 12

ShapeEditor – импорт и экспорт файлов фигур

В этой главе мы продолжим процесс реализации веб-приложения ShapeEditor. Мы начнем с создания режима просмотра данных в виде списка, который будет показывать имеющиеся файлы фигур, и затем разберем детали импорта и экспорта файлов фигур через веб-интерфейс.

В этой главе мы научимся:

- выводить на экран список записей, используя для этого шаблон Django;
- решать сложные проблемы, связанные с данными файла фигур, включая проблемы с геометриями и атрибутивными типами данных;
- импортировать данные файла фигур, используя для этого веб-интерфейс;
- экспортировать файл фигур, используя для этого тот же веб-интерфейс.

Реализация режима просмотра списка файлов фигур

Когда пользователи в самом начале открывают веб-приложение ShapeEditor, мы хотим, чтобы они увидели список ранее выгруженных на сервер файлов фигур с опциями импорта, правки, экспорта и удаления. Приложение веб-платформы Django с реализацией режима просмотра данных в виде списка и связанной с ним функциональностью будет называться `shapefiles` (файлы фигур); пойдем дальше и теперь создадим это приложение.

Откройте окно терминала или командной строки, перейдите при помощи команды `cd` в верхний каталог приложения `shapeEditor` и наберите следующую ниже команду:

```
python3 manage.py startapp shapefiles
```

Напомним еще раз, что эта команда создает приложение shapefiles на верхнем уровне, то есть приложение многократного использования. Переместите его вовнутрь каталога shapeEditor, набрав следующую команду:

```
mv shapefiles shapeEditor
```

Заодно перейдите в каталог shapeEditor/shapefiles и удалите модули admin.py и tests.py, поскольку они нам будут не нужны. Затем откройте модуль shapeEditor/settings.py в текстовом редакторе и добавьте в конце списка INSTALLED_APPS следующую ниже запись:

```
'shapeEditor.shapefiles',
```

Наше приложение shapefiles не задает своих собственных моделей базы данных, но нам нужно оставить файл models.py, с тем чтобы веб-платформа Django распознавала его как приложение. Еще одна интересная часть приложения – это модуль views.py, где мы определим наш режим просмотра данных «Вывести список файлов фигур» со списком файлов фигур. Пойдем дальше и напишем для этого режима простую заготовку; откройте модуль views.py в текстовом редакторе и введите туда следующий фрагмент:

```
from django.http import HttpResponse

def list_shapefiles (request):
    return HttpResponse("ответ из функции list_shapefiles")
```

В этой главе много исходного кода, но если вы не хотите, то вам не нужно набирать все это вручную. По своему усмотрению вы можете просто скачать пример программного кода к этой главе, который содержит все внесенные здесь изменения в систему ShapeEditor.

Далее нам нужно поручить нашему проекту ShapeEditor вызывать этот режим тогда, когда пользователь обращается к самому верхнему URL-адресу системы ShapeEditor. Для этого откройте модуль urls.py и отредактируйте его так, чтобы он выглядел, как показано ниже:

```
from django.conf.urls import include, url
from django.contrib.gis import admin
import shapeEditor.shapefiles.views

urlpatterns = [
    url(r'^$', shapeEditor.shapefiles.views.list_shapefiles),
    url(r'^admin/', include(admin.site.urls)),
]
```

Первый URL-шаблон преобразовывает самый верхний URL-адрес (который определяется регулярным выражением r'^$') в нашу просмотровую функцию shapeEditor.shapefiles.views.list_shapefiles(). Режим просмотра данных в виде списка выступает для пользователя в качестве стартовой точки во всей системе ShapeEditor. Второй определяемый нами URL-шаблон позволяет пользователю получать доступ к веб-интерфейсу администратора GeoDjango через URL-адрес /admin.

Пора теперь протестировать нашу новую просмотровую функцию. Наберите в окне терминала следующую ниже команду:

```
python3 manage.py runserver
```

Затем откройте веб-браузер и перейдите по следующему URL-адресу:

```
http://127.0.0.1:8000/
```

Если все выполнено правильно, то в окне браузера вы должны увидеть сообщение ответ из функции list_shapefiles. Оно говорит о том, что в ответ на URL-адрес верхнего уровня вызывается наша просмотровая функция list_shapefile().

Имея в распоряжении действующую просмотровую функцию, теперь заставим ее сделать что-то полезное. Откройте модуль views.py (в каталоге shapeEditor/shapefiles) и отредактируйте его содержимое так, чтобы оно выглядело, как показано ниже:

```
from django.http import HttpResponse
from django.shortcuts import render
from shapeEditor.shared.models import Shapefile

def list_shapefiles (request):
    shapefiles = Shapefile.objects.all().order_by('filename')
    return render(request, "list_shapefiles.html",
                  {'shapefiles' : shapefiles})
```

Просмотровая функция list_shapefiles() теперь делает две вещи:

- загружает список всех объектов Shapefile из базы данных в память, отсортированный по имени файла;
- передает этот список в шаблон веб-платформы Django (в файле с именем list_shapefiles.html), где он преобразуется в HTML-страницу, после чего результат возвращается вызывающей стороне.

Пойдем дальше и создадим шаблон list_shapefiles.html. Создайте каталог с именем templates внутри каталога shapeEditor/shapefiles и затем новый файл в этом каталоге под названием list_shapefiles.html. Этот файл должен иметь следующее ниже содержание:

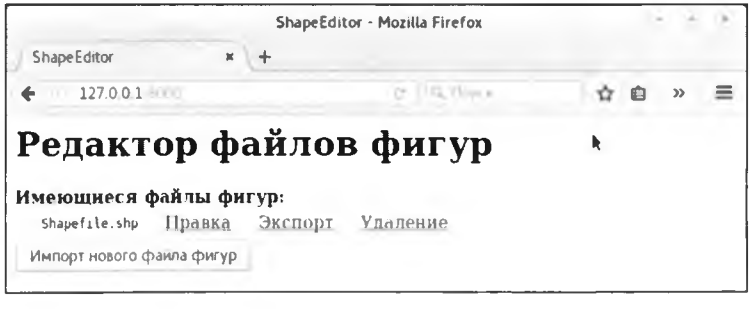
```
<html>
  <head>
    <title>ShapeEditor</title>
  </head>
  <body>
    <h1>Редактор файлов фигур</h1>
{% if shapefiles %}
    <b>Перечень имеющихся файлов фигур:</b>
    <table border="0" cellspacing="0" cellpadding="5"
           style="padding-left:20px">
  {% for shapefile in shapefiles %}
      <tr>
        <td><font style="font-family:monospace">
```

```
          {{ shapefile.filename }}
        </font></td>
        <td> </td>
        <td>
          <a href="/edit/{{ shapefile.id }}">Правка</a>
        </td>
        <td> </td>
        <td>
          <a href="/export/{{ shapefile.id }}">Экспорт</a>
        </td>
        <td> </td>
        <td>
          <a href="/delete/{{ shapefile.id }}">Удаление</a>
        </td>
      </tr>
  {% endfor %}
    </table>
{% endif %}
    <button type="button"
            onClick='window.location="/import";'>
      Импорт нового файла фигур
    </button>
  </body>
</html>
```

Этот шаблон работает следующим образом:

- если список shapefiles не пуст, то шаблон создает HTML-таблицу, чтобы показать содержимое списка;
- для каждой записи в списке shapefiles создается новая строка в таблице;
- каждая строка таблицы состоит из имени файла фигур (моноширинным шрифтом) и гиперссылок **Правка**, **Экспорт** и **Удаление**;
- наконец, в основание страницы выносится кнопка **Импорт нового файла фигур**.

Используемые в шаблоне ссылки мы рассмотрим чуть позже, а на данный момент просто создадим файл, удостоверимся, что сервер Django работает, и перезагрузим веб-браузер. Вы должны увидеть следующую страницу:



Как видите, на странице показан файл фигур, который был создан ранее в веб-интерфейсе администратора вместе с соответствующими гиперссылками и кнопкой для доступа к остальной части функциональности ShapeEditor:

- гиперссылка **Правка** направит пользователя на страницу с URL-адресом `/edit/1`, которая позволит пользователю отредактировать файл фигур с заданным идентификатором записи;
- гиперссылка **Экспорт** направит пользователя на страницу с URL-адресом `/export/1`, которая позволит пользователю скачать копию файла фигур с заданным идентификатором записи;
- гиперссылка **Удаление** направит пользователя на страницу с URL-адресом `/delete/1`, которая позволит пользователю удалить заданный файл фигур;
- кнопка **Импорт нового файла фигур** направит пользователя на страницу с URL-адресом `/import`, которая позволит пользователю выгрузить на сервер новый файл фигур.

При необходимости эти URL-адреса можно протестировать, нажав на соответствующих ссылках, – они всего лишь покажут страницу ошибки, но вы увидите, как эти URL-ссылки соединяют вместе различные функциональные компоненты приложения ShapeEditor. Можно также подробнее рассмотреть страницу ошибки Django, которая может оказаться довольно полезной в отслеживании программных ошибок.

Располагая действующей первой страницей, теперь начнем реализацию базовой функциональности системы ShapeEditor. Мы начнем с логики, необходимой для импорта файла фигур.

Импорт файлов фигур

Процесс импортирования файла фигур состоит из следующих этапов.

1. Вывести форму, предлагающую пользователю выгрузить на сервер архив ZIP с файлом фигур.
2. Распаковать zip-файл, чтобы извлечь выгруженный файл фигур.
3. Открыть файл фигур и считать его данные в базу данных.
4. Удалить временные файлы, созданные при выполнении предыдущих операций.

Разберем по очереди каждый из этих этапов.

Форма для импорта файлов фигур

Начнем с создания заготовки для режима просмотра «Импорт файла фигур». Откройте модуль `urls.py` в текстовом редакторе и добавьте новую запись в список `urlpatterns`:

```
url(r'^import$', shapeEditor.shapefiles.views.import_shapefile),
```

Затем откройте модуль `shapeEditor/shapefiles/views.py` и добавьте заготовку просмотровой функции `import_shapefile()`, которая будет выдавать ответ при запросе этого URL-адреса:

```
def import_shapefile(request):
    return HttpResponse("Продолжение следует...")
```

Если нужно, можно ее протестировать: запустите сервер Django, перейдите на главную страницу и щелкните по кнопке **Импорт нового файла фигур**. Вы должны увидеть сообщение **Продолжение следует...**.

Чтобы разрешить пользователю вводить данные, мы воспользуемся формой веб-платформы Django. Формы – это пользовательские классы; классы описывают различные поля, которые затем выводятся на веб-странице. В этом случае наша форма будет иметь одно-единственное поле, которое позволяет пользователю выбрать файл для выгрузки на сервер. Мы сохраним эту форму в файле с именем `forms.py` в каталоге `shapeEditor/shapefiles`. Пойдем дальше и создадим этот файл, а затем отредактируем его так, чтобы он выглядел следующим образом:

```
from django import forms

class ImportShapefileForm(forms.Form):
    import_file = forms.FileField(label="Выберите сжатый файл фигур")
```

Класс `FileField` позволяет пользователю выбрать файл для выгрузки. Как видите, мы даем этому полю пояснительную надпись, которая будет выведена на веб-странице.

Создав форму, теперь вернитесь к `views.py` и измените определение просмотровой функции `import_shapefile()`, чтобы оно выглядело следующим образом:

```
def import_shapefile(request):
    if request.method == "GET":
        form = ImportShapefileForm()
        return render(request, "import_shapefile.html",
                      {'form' : form})
    elif request.method == "POST":
        form = ImportShapefileForm(request.POST, request.FILES)

        if form.is_valid():
            shapefile = request.FILES['import_file']
            # Продолжение следует...
            return HttpResponseRedirect("/")

        return render(request, "import_shapefile.html",
                      {'form' : form})
```

Кроме того, добавьте этих двух операторов импорта в начало модуля:

```
from django.http import HttpResponseRedirect
from shapeEditor.shapefiles.forms import ImportShapefileForm
```

Проанализируем, что здесь происходит: вначале функция `import_shapefile()` вызывается с HTTP-запросом GET, в результате чего она создаст новый объект `ImportShapefileForm` и затем вызовет функцию `render()`, чтобы вывести эту форму

пользователю. Во время отправки формы функция import_shapefile() будет вызвана с HTTP-запросом POST. В этом случае объект ImportShapefileForm будет создан с отправляемыми данными (запросы request.POST и request.FILES), и форма будет проверена на допустимость вводимых данных. Если они допустимы, то мы извлечем выгруженный файл фигур.

Поскольку на данный момент фактический код импорта еще нами не реализован, мы просто перенаправим пользователя назад к странице верхнего уровня. Как говорится в комментарии, в этом месте мы вскоре внесем соответствующий программный код.

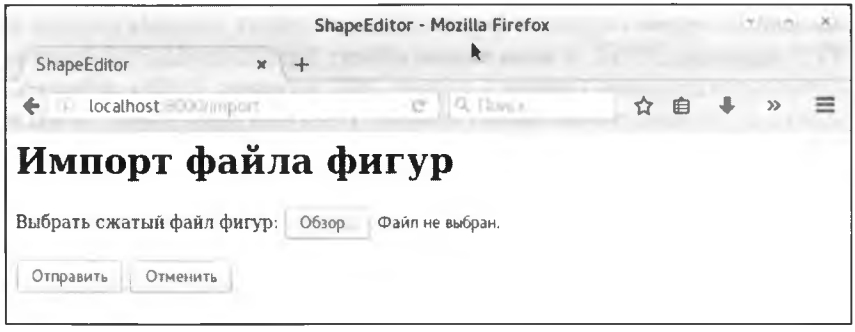
Если форма не была допустимой, мы еще раз вызовем render(), чтобы вывести форму пользователю. В этом случае Django автоматически выведет связанное с формой сообщение(я) об ошибке, чтобы пользователь увидел, почему проверка потерпела неудачу.

Чтобы вывести форму, мы воспользуемся шаблоном Django и передадим объект формы в качестве параметра. Теперь создадим этот шаблон, то есть новый файл под названием import_shapefile.html в каталоге templates, и введем туда следующий ниже текст:

```
<html>
  <head>
    <title>ShapeEditor</title>
  </head>
  <body>
    <h1>Импорт файла фигур</h1>
    <form enctype="multipart/form-data" method="post"
          action="import">
      {{ form.as_p }}
      <input type="submit" value="Отправить"/>
      <button type="button"
              onClick='window.location="/shape-editor";'>
              Отменить
      </button>
    </form>
  </body>
</html>
```

Как видите, этот шаблон определяет элемент HTML <form> и добавляет в эту форму кнопки **Отправить** и **Отменить**. Тело формы не определено. Вместо этого мы используем {{ form.as_p }}, чтобы вставить объект формы в шаблон.

Проверим его работу. Запустите веб-сервер Django, в случае если он еще не работает, откройте веб-браузер и в строку поиска введите URL-адрес http://127.0.0.1:8000. Затем щелкните по кнопке **Импорт нового файла фигур**. Если все выполнено правильно, то вы должны увидеть следующую ниже страницу:



При попытке отправить форму без выгрузки данных на сервер появится сообщение об ошибке, говорящее, что требуется файл фигур. Эта обработка ошибок выполняется по умолчанию для любой формы: по умолчанию требуется наличие значений всех полей. Если же вы выберете файл для выгрузки, то пользователь всего лишь будет перенаправлен назад к главной странице; это объясняется тем, что мы еще не реализовали логику программы, связанную с импортом.

После создания формы теперь займемся программным кодом, который требуется для обработки выгруженного файла фигур.

Извлечение выгруженного файла фигур из архива

Поскольку процесс импорта данных будет относительно трудоемким, мы поместим этот программный код в отдельный модуль. Создайте внутри каталога `shapeEditor/shapefiles` новый файл с именем `shapefileio.py` и добавьте в этот файл следующий ниже текст:

```
def import_data(shapefile):
    return "Продолжение следует..."
```

Учитывая, что процесс импорта файла фигур вполне может потерпеть неудачу (например, если пользователь выгружает файл, который не является архивом ZIP), наша функция `import_data()`, в случае если что-то пойдет не так, вернет сообщение об ошибке. Вернемся и изменим наш режим просмотра данных (и шаблон), чтобы вызвать эту функцию `import_data()` и вывести возвращенное сообщение об ошибке, если таковое имеется.

Откройте модуль `views.py` в текстовом редакторе и добавьте в просмотровую функцию `import_shapefile()` следующие ниже выделенные строки:

```
def import_shapefile(request):
    if request.method == "GET":
        form = ImportShapefileForm()
        return render_to_response("import_shapefile.html",
                                  {'form' : form,
                                   'errMsg' : None})
    elif request.method == "POST":
        errMsg = None # исходно.
```

```
    form = ImportShapefileForm(request.POST, request.FILES)

    if form.is_valid():
        shapefile = request.FILES['import_file']
        errMsg = shapefileIO.import_data(shapefile)
        if errMsg == None:
            return HttpResponseRedirect("/")

    return render_to_response("import_shapefile.html",
                              {'form' : form,
                               'errMsg' : errMsg})
```

Вам также в самом начале файла нужно добавить нижеследующие строки:

```
from django.shortcuts import render_to_response
from shapeEditor.shapefiles import shapefileIO
```

Далее откройте в текстовом редакторе шаблон `import_shapefile.html` и сразу после строки `<h1>Import Shapefile<h1>` добавьте в файл следующие ниже строки:

```
{% if errMsg %}
<b><i>{{ errMsg }}</i></b>
{% endif %}
```

В результате пользователю будет выведено сообщение об ошибке, в случае если оно не равно `None`.

Пойдем дальше и опробуем свои изменения: щелкните по кнопке **Импорт нового файла фигур**, выберите архив ZIP и щелкните по **Отправить**. Вы должны увидеть, что вверху формы появится текст **Продолжение следует...**, то есть текст ошибки из нашей заготовки функции `import_data()`.

Теперь мы готовы приступить к реализации логики импорта файлов фигур. Снова откройте в текстовом редакторе модуль `shapefileIO.py` и приготовьтесь написать тело функции `import_data()`. Мы пройдем этот этап шаг за шагом.

Когда мы настраиваем форму, которая содержит объект `FileField`, веб-платформа Django возвращает нам объект `UploadedFile`, представляющий выгруженный файл. Наша первая задача состоит в том, чтобы прочитать содержимое объекта `UploadedFile` и сохранить его на диске во временном файле, чтобы мы могли с ним работать. Добавьте к своей функции `import_data()` следующий фрагмент:

```
fd,fname = tempfile.mkstemp(suffix=".zip")
os.close(fd)

f = open(fname, "wb")
for chunk in shapefile.chunks():
    f.write(chunk)
f.close()
```

Как видите, чтобы создать временный файл, мы используем модуль `tempfile` из стандартной библиотеки Python и затем копируем содержимое объекта `shapefile` в этот файл.

Поскольку `tempfile.mkstemp()` возвращает дескриптор и имя файла, мы вызываем `os.close(fd)`, чтобы закрыть дескриптор файла. Это позволяет нам, следуя нормальной процедуре, открыть файл вновь, используя для этого `open()`, и выполнить в него запись.

Теперь мы готовы открыть временный файл и проверить, является ли он действительно архивом ZIP, содержащим файлы, которые составляют файл фигур. Вот как это можно сделать:

```
if not zipfile.is_zipfile(fname):
    os.remove(fname)
    return "Недопустимый архив ZIP."

zip = zipfile.ZipFile(fname)

required_suffixes = [".shp", ".shx", ".dbf", ".prj"]
has_suffix = {}
for suffix in required_suffixes:
    has_suffix[suffix] = False

for info in zip.infolist():
    suffix = os.path.splitext(info.filename)[1].lower()
    if suffix in required_suffixes:
        has_suffix[suffix] = True

for suffix in required_suffixes:
    if not has_suffix[suffix]:
        zip.close()
        os.remove(fname)
        return "В архиве отсутствует необходимый " + suffix + " файл."
```

Отметим, что для проверки содержимого выгруженного архива ZIP мы используем модуль `zipfile` стандартной библиотеки Python и возвращаем подходящее сообщение об ошибке, в случае если что-то не так. И, прежде чем возвратить сообщение об ошибке, мы также удаляем временный файл, чтобы временные файлы не накапливались.

Наконец, зная, что выгруженный файл – это допустимый архив ZIP, содержащий файлы, которые составляют файл фигур, теперь мы можем извлечь эти файлы и сохранить их во временном каталоге:

```
shapefile_name = None
dir_name = tempfile.mkdtemp()
for info in zip.infolist():
    if info.filename.endswith(".shp"):
        shapefile_name = info.filename

    dst_file = os.path.join(dir_name, info.filename)
    f = open(dst_file, "wb")
    f.write(zip.read(info.filename))
    f.close()
zip.close()
```

Отметим, что, прежде чем скопировать извлеченные файлы во временный каталог, мы сначала его создаем. В то же время мы сохраняем имя главного .shp файла архива, так как оно нам пригодится, когда мы откроем файл фигур.

Поскольку в этом фрагменте программного кода мы воспользовались некоторыми модулями стандартной библиотеки Python, вам также придется добавить в начало модуля следующую ниже строку:

```
import os, os.path, tempfile, zipfile
```

Импорт содержимого файла фигур

После извлечения файлов из архива ZIP теперь мы готовы импортировать данные из выгруженного файла фигур. Процесс импорта содержимого файла фигур состоит из следующих этапов.

1. Открыть файл фигур.
2. Добавить объект Shapefile к базе данных.
3. Определить атрибуты объекта Shapefile.
4. Сохранить геообъекты файла фигур в объекте Shapefile.
5. Сохранить атрибуты файла фигур в объекте Shapefile.

Разберем по очереди каждый из этих этапов.

Открытие файла фигур

Для открытия файла фигур мы воспользуемся библиотекой OGR. Добавьте в конец своего модуля shapefileIO.py следующий ниже фрагмент:

```
try:
    datasource = ogr.Open(os.path.join(dir_name, shapefile_name))
    layer = datasource.GetLayer(0)
    shapefile_ok = True

except:
    traceback.print_exc()
    shapefile_ok = False

if not shapefile_ok:
    os.remove(fname)
    shutil.rmtree(dirname)
    return "Недопустимый файл фигур."
```

И снова, если что-то идет не так, мы удаляем наши временные файлы и возвращаем подходящее сообщение об ошибке. Мы также используем библиотечный модуль traceback, который выводит отладочную информацию в журнал веб-сервера, возвращая при этом дружественное сообщение об ошибке, которое будет показано пользователю.

В этой программе для чтения и записи файлов фигур мы будем использовать библиотеку OGR напрямую. Модуль GeoDjango предлагает для библиотеки OGR свой собственный питоновский интерфейс, который реализован в пакете contrib.gis.gdal, но, к сожалению, в версии GeoDjango запись в файлы фигур не реализована.

В этой связи мы воспользуемся непосредственно привязками Python для библиотеки OGR, которая требует, чтобы библиотека OGR была установлена отдельно.

Поскольку в этом фрагменте используется несколько модулей стандартной библиотеки Python, а также библиотека OGR, в начало модуля shapefileIO.py нам придется добавить следующие ниже операторы импорта:

```
import shutil, traceback
from osgeo import ogr
```

Добавление объекта Shapefile в базу данных

Успешно открыв файл фигур, теперь мы готовы прочитать из него данные. Прежде всего мы создадим объект Shapefile, который будет представлять этот импортированный файл фигур. Для этого добавьте в конец своей функции import_data() следующий ниже фрагмент кода:

```
src_spatial_ref = layer.GetSpatialRef()
geom_type = layer.GetLayerDefn().GetGeomType()
geom_name = ogr.GeometryTypeToName(geom_type)
shapefile = Shapefile(filename=shapefile_name,
                      srs_wkt=src_spatial_ref.ExportToWkt(),
                      geom_type=geom_name)
shapefile.save()
```

Как видите, мы получаем из слоя файла фигур пространственную привязку, а также имя типа хранящейся в этом слое геометрии. Затем мы сохраняем имя файла фигур, пространственную привязку и имя типа геометрии в объекте Shapefile, после чего сохраняем этот объект в базе данных.

Чтобы этот программный код мог работать, в начало модуля shapefileIO.py мы должны добавить следующий ниже оператор импорта:

```
from shapeEditor.shared.models import Shapefile
```

Определение атрибутов объекта Shapefile

Создав объект Shapefile, который представляет импортированный файл фигур, наша следующая задача теперь состоит в том, чтобы создать объекты Attribute, описывающие атрибуты файла фигур. Это можно сделать, опросив файл фигур при помощи библиотеки OGR; добавьте в конец функции import_data() следующий ниже фрагмент:

```
attributes = []
layer_def = layer.GetLayerDefn()
for i in range(layer_def.GetFieldCount()):
    field_def = layer_def.GetFieldDefn(i)
    attr = Attribute(shapefile=shapefile,
                     name=field_def.GetName(),
                     type=field_def.GetType(),
                     width=field_def.GetWidth(),
```

```
                precision=field_def.GetPrecision())
    attr.save()
    attributes.append(attr)
```

Отметим, что наряду с сохранением объектов `Attribute` в базе данных мы также создаем отдельный список этих атрибутов в переменной `attributes`. Мы воспользуемся им позже, когда мы будем импортировать значения атрибутов для каждого геообъекта.

Напомним, что следует добавить в начало модуля следующий ниже оператор импорта:

```
from shapeEditor.shared.models import Attribute
```

Извлечение и сохранение геообъектов

Наша следующая задача состоит в том, чтобы извлечь геообъекты файла фигур и сохранить их в качестве объектов `Feature` в базе данных. Поскольку геообъекты файла фигур могут быть в любой пространственной привязке, мы должны перед их сохранением трансформировать их в нашу внутреннюю систему пространственной привязки (EPSG-код 4326, неспроецированные значения широты и долготы). Для этого мы воспользуемся объектом трансформации координат `CoordinateTransformation` библиотеки `OGR`.

При помощи библиотеки `OGR` можно легко просмотреть все геообъекты файла фигур, извлечь геометрию каждого геообъекта, трансформировать их в пространственную привязку с EPSG-кодом 4326 и конвертировать их в объект геометрии GEOS модуля GeoDjango, с тем чтобы его сохранить в базе данных. К сожалению, эта работа ставит перед нами две задачи, которые мы должны решить:

- как мы уже убедились в предыдущей главе, файлы фигур не способны отличить многоугольники от составных многоугольников, ломаные от составных ломаных и точки от составных точек. Если файл фигур содержит многоугольники, то некоторые геообъекты могут содержать геометрию многоугольника, в то время как другие – геометрию составного многоугольника. Поскольку все геообъекты в базе данных должны иметь одинаковый тип геометрии, нам придется *завертывать* геометрию многоугольника в геометрию составного многоугольника, геометрию ломаной в геометрию составной ломаной и геометрию точки в геометрию составной точки, чтобы они гарантированно имели одинаковый тип геометрии;
- поскольку объект `Feature` имеет отдельные поля для каждого типа геометрии, мы должны решить, какое конкретное поле внутри объекта `Feature` будет содержать заданную геометрию. Когда мы определяли класс `Feature`, нам пришлось создать отдельные поля геометрии для каждого типа геометрии; теперь нам нужно решить, какое из этих полей использовать для хранения заданного типа геометрии.

Учитывая, что эти две задачи (завертывание геометрии и выбор поля геометрии, используемого для заданного типа геометрии) являются функциями общего вида, которые нам придется использовать в разных местах, мы сохраним их в отдельном

модуле. В каталоге shapeEditor/shared создайте файл с именем utils.py и затем добавьте в этот модуль нижеследующий программный код:

```
from django.contrib.gis.geos.collections \
import MultiPolygon, MultiLineString, MultiPoint

def wrap_geos_geometry(geometry):
    if geometry.geom_type == "Polygon":
        return MultiPolygon(geometry)
    elif geometry.geom_type == "LineString":
        return MultiLineString(geometry)
    elif geometry.geom_type == "Point":
        return MultiPoint(geometry)
    else:
        return geometry

def calc_geometry_field(geometry_type):
    if geometry_type == "Polygon":
        return "geom_multipolygon"
    elif geometry_type == "LineString":
        return "geom_multilinestring"
    elif geometry_type == "Point":
        return "geom_multipoint"
    else:
        return "geom_" + geometryType.lower()
```

Каждая приличная программа Python должна иметь модуль utils.py; самое время его добавить к приложению ShapeEditor.

Располагая этим модулем, мы теперь можем вернуться к нашему модулю shapefileIO.py и завершить реализацию функции import_data(). Чтобы сохранить геообъекты на диск, мы сначала должны настроить объект трансформации координат, который конвертирует наши геообъекты из системы пространственной привязки файла фигур во внутреннюю с EPSG-кодом 4326. Добавьте в конец своей функции import_data() следующий ниже фрагмент программы:

```
dst_spatial_ref = osr.SpatialReference()
dst_spatial_ref.ImportFromEPSG(4326)

coord_transform = osr.CoordinateTransformation(
                                    src_spatial_ref,
                                    dst_spatial_ref)
```

Теперь мы можем в итеративном режиме просканировать геообъекты файла фигур, создавая для каждого геообъекта объект GeoDjango GEOSGeometry:

```
for i in range(layer.GetFeatureCount()):
    src_feature = layer.GetFeature(i)
    src_geometry = src_feature.GetGeometryRef()
    src_geometry.Transform(coord_transform)
    geometry = GEOSGeometry(src_geometry.ExportToWkt())
    geometry = utils.wrap_geos_geometry(geometry)
```

Отметим, что мы используем написанную нами ранее функцию `utils.wrap_geos_geometry()` для завертывания геометрии многоугольника, ломаной и точки, чтобы их можно было сохранить в базе данных.

Теперь нам нужно сохранить геометрию внутри объекта `Feature`. Для этого мы сначала вычислим имя поля геометрии, которое будет содержать нашу геометрию:

```
geom_field = utils.calc_geometry_field(geom_name)
```

Зная имя поля, теперь можно создать объект `Feature`, в соответствующем поле которого будет сохранена геометрия. Поскольку имя поля варьируется, мы воспользуемся приемом Python под названием *распаковка именованных аргументов*. Он подразумевает создание словаря Python с различными полями, которые мы хотим сохранить в объекте `Feature`:

```
fields = {}
fields['shapefile'] = shapefile
fields[geom_field] = geometry
```

Затем мы воспользуемся этим словарем как набором именованных параметров для передачи инициализатору объекта `Feature`. Это приведет к тому, что будет создан новый объект `Feature` с заданными полями, который мы затем сохраним в базе данных:

```
feature = Feature(**fields)
feature.save()
```

Наконец, добавим в начало модуля `shapefileIO.py` следующие ниже операторы импорта `import`:

```
from django.contrib.gis.geos.geometry import GEOSGeometry
from osgeo import osr
from shapeEditor.shared.models import Feature
from shapeEditor.shared import utils
```

Извлечение и сохранение атрибутов

Разобравшись с геометрией геообъекта, мы теперь можем обратиться к импорту атрибутов геообъекта. В своей основе процесс состоит из итеративного просмотра атрибутов, извлечения каждого значения атрибута из геообъекта при помощи библиотеки `OGR`, создания объекта `AttributeValue`, в котором будет содержаться его значение, и затем сохранения атрибута в базе данных. Наш программный код будет выглядеть следующим образом:

```
for attr in attributes:
    value = ...
    attr_value = AttributeValue(feature=feature,
                                attribute=attr,
                                value=value)
    attr_value.save()
```

Проблема заключается в том, как извлечь значение атрибута из геообъекта. Поскольку объект Feature библиотеки OGR для разных типов значений полей имеет разные методы, нам придется выполнять проверку на разные типы полей, вызывать соответствующий метод GetFieldAsxxx(), конвертировать получающееся значение в строку и затем сохранять эту строку в объекте AttributeValue. При этом значения Null тоже следует обработать соответствующим образом. Ввиду всех этих сложностей мы определим внутри utils.py новую функцию, которая будет проделывать все тяжелую работу, и будем просто вызывать ее из нашей функции import_data(). Назовем эту функцию get_ogr_feature_attribute().

Отметим, что нам придется извлекать значения атрибутов из файла и выполнять их разбор, вследствие чего процесс извлечения значения атрибута может фактически потерпеть неудачу, например в случае, если файл фигур содержал тип атрибута, который не поддерживается. В связи с этим нам нужно добавить в наш код обработку ошибок. Для поддержания обработки ошибок наша функция get_ogr_feature_attribute() будет возвращать кортеж в формате (success, result), где success будет True, если и только если атрибут был извлечен успешно, а result будет содержать либо значение извлеченного атрибута (в виде последовательности символов), либо сообщение об ошибке, объясняющее, почему операция потерпела неудачу.

Добавим в нашу функцию import_data() необходимый программный код сохранения значений атрибутов в базе данных и корректной обработки любых ошибок конвертации, которые могут произойти:

```
for attr in attributes:
    success,result = utils.get_ogr_feature_attribute(
                              attr, src_feature)
    if not success:
        os.remove(fname)
        shutil.rmtree(dir_name)
        shapefile.delete()
        return result

    attr_value = AttributeValue(feature=feature,
                                attribute=attr,
                                value=result)
    attr_value.save()
```

Отметим, что в функцию get_ogr_ feature_attribute() мы передаем объект Attribute и геообъект библиотеки OGR. Если происходит ошибка, мы удаляем временные файлы, удаляем созданный ранее файл фигур и возвращаем вызывающей стороне сообщение об ошибке. Если атрибут был успешно извлечен, мы создаем новый объект AttributeValue со значением атрибута и сохраняем его в базе данных.

Отметим, что мы используем shapefile.delete(), чтобы удалить из базы данных частично импортированный файл фигур. По умолчанию веб-платформа Django также автоматически удалит любые записи, которые связаны с записью, удаляемой через поле ForeignKey. Это означает, что объект Shapefile будет удален наряду со

всеми связанными с ним объектами Attribute, Feature **и** AttributeValue. **При помощи всего одной строки кода мы можем полностью удалить все ссылки на данные, принадлежащие файлу фигур.**

Теперь реализуем функцию get_ogr_feature_attribute(). Добавьте в utils.py следующий ниже программный код:

```
def get_ogr_feature_attribute(attr, feature):
    attr_name = attr.name

    if not feature.IsFieldSet(attr_name):
        return (True, None)

    if attr.type == ogr.OFTInteger:
        value = str(feature.GetFieldAsInteger(attr_name))
    elif attr.type == ogr.OFTIntegerList:
        value = repr(feature.GetFieldAsIntegerList(attr_name))
    elif attr.type == ogr.OFTReal:
        value = feature.GetFieldAsDouble(attr_name)
        value = "%*.*f" % (attr.width, attr.precision, value)
    elif attr.type == ogr.OFTRealList:
        values = feature.GetFieldAsDoubleList(attr_name)
        str_values = []
        for value in values:
            str_values.append("%*.*f" % (attr.width,
                                         attr.precision, value))
        value = repr(str_values)
    elif attr.type == ogr.OFTString:
        value = feature.GetFieldAsString(attr_name)
    elif attr.type == ogr.OFTStringList:
        value = repr(feature.GetFieldAsStringList(attr_name))
    elif attr.type == ogr.OFTDate:
        parts = feature.GetFieldAsDateTime(attr_name)
        year,month,day,hour,minute,second,tzone = parts
        value = "%d,%d,%d,%d" % (year,month,day,tzone)
    elif attr.type == ogr.OFTTime:
        parts = feature.GetFieldAsDateTime(attr_name)
        year,month,day,hour,minute,second,tzone = parts
        value = "%d,%d,%d,%d" % (hour,minute,second,tzone)
    elif attr.type == ogr.OFTDateTime:
        parts = feature.GetFieldAsDateTime(attr_name)
        year,month,day,hour,minute,second,tzone = parts
        value = "%d,%d,%d,%d,%d,%d,%d,%d" % (year,month,day,
                                             hour,minute,
                                             second,tzone)
    # учтем 64-разрядные значения
    elif attr.type == ogr.OFTInteger64:
        value = str(feature.GetFieldAsInteger64(attr_name))
    elif attr.type == ogr.OFTIntegerList64:
        value = repr(feature.GetFieldAsIntegerList64(attr_name))
```

```
    else:
        return (False, "Тип атрибута не поддерживается: " +
                       str(attr.type))

    return (True, value)
```

Здесь функция содержит много невзрачного кода, касающегося извлечения различных типов полей из геообъекта библиотеки OGR. Не слишком переживайте по поводу деталей его реализации; в сущности, мы извлекаем значение атрибута в любом формате, который требуется для этого типа атрибута, при необходимости конвертируем значение(я) в последовательность символов и затем возвращаем ее вызывающей стороне.

Кроме того, для того чтобы новая функция заработала, нам также нужно добавить в начало utils.py следующую ниже строку программного кода:

```
from osgeo import ogr
```

Наконец, нам нужно добавить в начало модуля shapefileIO.py следующий ниже оператор импорта:

```
from shapeEditor.shared.models import AttributeValue
```

Очистка

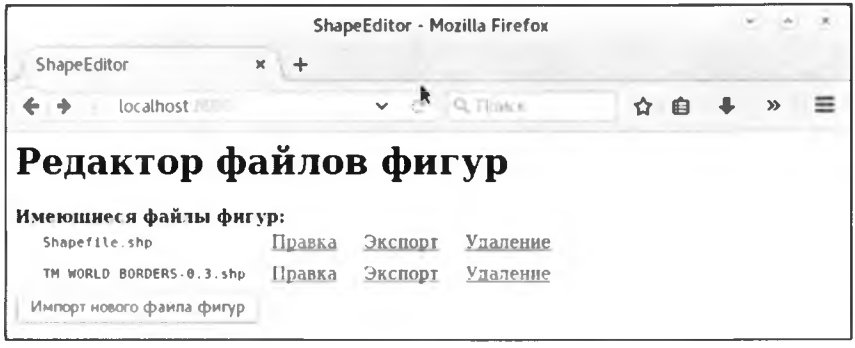
Импортировав данные файла фигур, теперь осталось всего лишь удалить наши временные файлы и сообщить вызывающей стороне, что импорт был выполнен успешно. Для этого просто добавьте в конец своей функции import_data() следующие ниже строки:

```
os.remove(fname)
shutil.rmtree(dir_name)
return None
```

Вот и все!

Чтобы все это перепроверить, возьмите копию файла фигур TM_WORLD_BORDERS-0.3 в формате zip-файла. Можно воспользоваться исходным архивом ZIP, который вы скачали с веб-сайта набора данных границ стран мира, либо повторно сжать файл фигур в новый архив ZIP. Затем запустите веб-приложение ShapeEditor, щелкните по кнопке **Импорт нового файла фигур**, щелкните по соответствующей кнопке, чтобы выбрать файл для импорта, и выберите архив ZIP с набором данных границ стран мира.

Когда вы щелкнете по кнопке **Отправить**, вам придется подождать несколько секунд, пока файл фигур будет импортирован. Если все выполнено правильно, набор данных границ стран мира появится в списке импортированных файлов фигур:



Если возникла проблема, проверьте сообщение об ошибке, чтобы понять, что стало причиной ее появления. Кроме того, вернитесь и удостоверьтесь, что вы набрали программный код в точности, как описано. Если же он работает, то примите наши поздравления! Вы только что реализовали самую трудную часть системы ShapeEditor. С этого места все станет легче.

Экспорт файлов фигур

Далее нам нужно реализовать возможность экспортировать файл фигур. Процесс экспорта файла фигур, в сущности, обратный логике импорта и состоит из следующих этапов.

1. Создать файл фигур библиотеки OGR, чтобы принять экспортируемые данные.
2. Сохранить объекты и их атрибуты в файле фигур.
3. Сжать файл фигур в архив ZIP.
4. Удалить временные файлы.
5. Передать zip-файл назад в веб-браузер пользователя.

Вся эта работа будет происходить в модуле `shapefileio.py` при помощи нескольких функций из `utils.py`. Вначале определим функцию `export_data()`, чтобы у нас было место, куда поместить наш программный код. Откройте `shapefileIO.py` в текстовом редакторе и добавьте следующую ниже новую функцию:

```
def export_data(shapefile):
    return "Продолжение следует..."
```

Заодно создадим просмотровую функцию «Экспорт файла фигур». Она будет вызывать функцию `export_data()`, которая выполнит всю работу. Откройте файл `shapefiles/views.py` и добавьте следующую ниже новую функцию:

```
def export_shapefile(request, shapefile_id):
    try:
        shapefile = Shapefile.objects.get(id=shapefile_id)
    except Shapefile.DoesNotExist:
        raise Http404("Такой файл фигур не существует")

    return shapefileIO.export_data(shapefile)
```

Все выглядит довольно прямолинейно: мы загружаем объект Shapefile с заданным идентификатором из базы данных и передаем его нашей функции export_data(). Открыв этот файл в текстовом редакторе, добавьте в его начало следующий ниже оператор импорта:

```
from django.http import Http404
```

Далее нам нужно поручить веб-платформе Django вызывать эту просмотровую функцию в ответ на URL-адрес /export. Откройте файл urls.py (в главном каталоге системы shapeEditor) и добавьте в список URL-шаблонов следующую ниже запись:

```
url(r'^export/(?P<shapefile_id>\d+)$',
        shapeEditor.shapefiles.views.export_shapefile),
```

Режим просмотра данных «Вывести список файлов фигур» обратится к этому URL-адресу, когда пользователь щелкнет по гиперссылке **Экспорт**. Она, в свою очередь, вызовет нашу просмотровую функцию, которая вызовет shapefileIO.export_data(), чтобы экспортировать файл фигур.

Теперь начнем реализовывать функцию export_data().

Определение файла фигур средствами OGR

Мы воспользуемся библиотекой OGR, чтобы создать новый файл фигур, в котором будут содержаться экспортируемые геообъекты. Начнем с создания временного каталога для содержимого файла фигур; замените свою версию заготовки функции export_data() следующим фрагментом программного кода:

```
def export_data(shapefile):
    dst_dir = tempfile.mkdtemp()
    dst_file = os.path.join(dst_dir, shapefile.filename)
```

Располагая местом, где можно хранить файл фигур (и именем файла для него), теперь создадим применяемую в файле фигур пространственную привязку, а также настроим для него источник данных и слой:

```
    dst_spatial_ref = osr.SpatialReference()
    dst_spatial_ref.ImportFromWkt(shapefile.srs_wkt)

    driver = ogr.GetDriverByName("ESRI Shapefile")
    datasource = driver.CreateDataSource(dst_file)
    layer = datasource.CreateLayer(shapefile.filename,
                                   dst_spatial_ref)
```

Создав непосредственно сам файл фигур, теперь нам нужно определить различные поля, в которых будут содержаться атрибуты файла фигур:

```
    for attr in shapefile.attribute_set.all():
        field = ogr.FieldDefn(attr.name, attr.type)
        field.SetWidth(attr.width)
        field.SetPrecision(attr.precision)
        layer.CreateField(field)
```

Отметим то, как информация, необходимая для определения поля, извлекается непосредственно из объекта Attribute: веб-платформа Django упрощает итерацию по атрибутам файла фигур.

На этом определение непосредственно самого файла фигур завершено. Теперь мы готовы начать сохранять геообъекты в файле фигур.

Сохранение геообъектов в файле фигур

Поскольку в файле фигур может использоваться любая допустимая пространственная привязка, мы должны трансформировать ее геообъекты из пространственной привязки, используемой для внутренних целей (EPSG 4326), в собственную пространственную привязку файла фигур. Для этого нам нужно создать объект трансформации координат osr.CoordinateTransformation, который выполнит трансформирование:

```
src_spatial_ref = osr.SpatialReference()
src_spatial_ref.ImportFromEPSG(4326)

coord_transform = osr.CoordinateTransformation(
                    src_spatial_ref, dst_spatial_ref)
```

Нам также нужно узнать, какое поле геометрии в объекте Feature содержит данные геометрии геообъекта:

```
geom_field = utils.calc_geometry_field(shapefile.geom_type)
```

Имея в распоряжении эту информацию, мы готовы начать экспортировать геообъекты в файл фигур:

```
for feature in shapefile.feature_set.all():
    geometry = getattr(feature, geom_field)
```

Однако прямо сейчас у нас есть небольшая проблема. Если вы помните, когда мы импортировали файл фигур, нам нужно было *завернуть* геометрию многоугольника, ломаной и точки в геометрию составного многоугольника, составной ломаной и составной точки, чтобы типы геометрии были согласованы с базой данных. Экспортируя файл фигур, мы должны *развернуть* геометрию так, чтобы геообъекты, в чьих геометриях имеется всего один многоугольник, ломаная или точка, сохранялись как многоугольники, ломаные и точки, а не как составные многоугольники, ломаные и точки. Для того чтобы выполнить это развертывание, мы воспользуемся функцией из utils.py:

```
    geometry = utils.unwrap_geos_geometry(geometry)
```

Чуть позже мы реализуем эту функцию utils.py.

Развернув геометрию геообъекта, теперь можно пойти дальше и конвертировать ее назад в геометрию библиотеки OGR, трансформировать ее в собственную для файла фигур систему пространственной привязки и затем, используя эту геометрию, создать геообъект OGR:

```
        dst_geometry = ogr.CreateGeometryFromWkt(geometry.wkt)
        dst_geometry.Transform(coord_transform)
        dst_feature = ogr.Feature(layer.GetLayerDefn())
        dst_feature.SetGeometry(dst_geometry)
```

Далее нам нужно добавить геообъект к слою, чтобы его можно было сохранить в файле фигур:

```
        layer.CreateFeature(dst_feature)
```

Наконец, нам нужно закрыть файл фигур, чтобы все данные гарантированно были сохранены на диске. Библиотека OGR не располагает явной функцией «закрытия» файла; вместо этого нам придется удалить наши ссылки на слой и источник данных. В результате это приведет к тому, что файл фигур будет закрыт и содержимое файла фигур будет записано на диск:

```
    layer = None
    datasource = None
```

Прежде чем мы пойдем дальше, добавим в `utils.py` нашу новую функцию `unwrap_geos_geometry()`. Ее программный код довольно прямолинейный: если объект составного многоугольника MultiPolygon, составной ломаной MultiLineString или составной точки MultiPoint содержит всего одну геометрию, то функция извлекает из него единственный объект – многоугольник, ломаную или точку:

```
def unwrap_geos_geometry(geometry):
    if geometry.geom_type in ["MultiPolygon",
                              "MultiLineString",
                              "MultiPoint"]:
        if len(geometry) == 1:
            geometry = geometry[0]
    return geometry
```

Пока неплохо: мы создали геообъект на основе библиотеки OGR, развернули его геометрию и сохранили все данные в файле фигур. Теперь мы готовы сохранить значения атрибутов геообъекта.

Сохранение атрибутов в файле фигур

Наша следующая задача состоит в том, чтобы сохранить значения атрибутов, связанных с каждым геообъектом. Когда мы импортировали файл фигур, мы извлекали значения атрибутов из различных типов данных библиотеки `OGR` и конвертировали их в последовательности символов, чтобы их можно было сохранить в базе данных. Это было сделано при помощи функции `utils.get_ogr_feature_attribute()`. Теперь нам нужно выполнить противоположное: сохранить строковое значение назад в поле атрибута библиотеки `OGR`. Как и прежде, для этого мы воспользуемся функцией из `utils.py`, которая проделает всю тяжелую работу. Добавьте в конец вашей функции `export_data()` из модуля `shapefileIO.py` следующие ниже выделенные строки:

```
        ...
        dst_feature = ogr.Feature(layer.GetLayerDefn())
        dst_feature.SetGeometry(dst_geometry)

        for attr_value in feature.attributevalue_set.all():
            utils.set_ogr_feature_attribute(
                    attr_value.attribute,
                    attr_value.value,
                    dst_feature)
        layer.CreateFeature(dst_feature)

    layer = None
    datasource = None
```

Теперь займемся реализацией функции set_ogr_feature_attribute() внутри utils.py. Как и в случае с функцией get_ogr_feature_attribute(), функция set_ogr_feature_attribute() довольно утомительная, но прямолинейная: нам нужно разобрать все типы данных OGR по очереди, обрабатывая строковое представление значения атрибута и вызывая соответствующий метод SetFieldXXX(), чтобы установить значение поля. Ниже приведен соответствующий программный код:

```
def set_ogr_feature_attribute(attr, value, feature):
    attr_name = attr.name

    if value == None:
        feature.UnsetField(attr_name)
        return

    if attr.type == ogr.OFTInteger:
        feature.SetField(attr_name, int(value))
    elif attr.type == ogr.OFTIntegerList:
        integers = eval(value)
        feature.SetFieldIntegerList(attr_name, integers)
    elif attr.type == ogr.OFTReal:
        feature.SetField(attr_name, float(value))
    elif attr.type == ogr.OFTRealList:
        floats = []
        for s in eval(value):
            floats.append(eval(s))
        feature.SetFieldDoubleList(attr_name, floats)
    elif attr.type == ogr.OFTString:
        feature.SetField(attr_name, value)
    elif attr.type == ogr.OFTStringList:
        strings = []
        for s in eval(value):
            strings.append(s.encode(encoding))
        feature.SetFieldStringList(attr_name, strings)
    elif attr.type == ogr.OFTDate:
        parts = value.split(",")
        year = int(parts[0])
```

```
        month = int(parts[1])
        day = int(parts[2])
        tzone = int(parts[3])
        feature.SetField(attr_name, year, month, day,
                         0, 0, 0, tzone)
    elif attr.type == ogr.OFTTime:
        parts = value.split(",")
        hour = int(parts[0])
        minute = int(parts[1])
        second = int(parts[2])
        tzone = int(parts[3])
        feature.SetField(attr_name, 0, 0, 0,
                         hour, minute, second, tzone)
    elif attr.type == ogr.OFTDateTime:
        parts = value.split(",")
        year = int(parts[0])
        month = int(parts[1])
        day = int(parts[2])
        hour = int(parts[3])
        minute = int(parts[4])
        second = int(parts[5])
        tzone = int(parts[6])
        feature.SetField(attr_mame, year, month, day,
                         hour, minute, second, tzone)
```

Сжатие файла фигур

Выполнив экспорт нужных данных в файл фигур библиотеки OGR, теперь можно их сжать в архив ZIP. Вернитесь в модуль shapefileIO.py и добавьте в конец своей функции export_data() следующий ниже фрагмент:

```
temp = tempfile.TemporaryFile()
zip = zipfile.ZipFile(temp, 'w', zipfile.ZIP_DEFLATED)

shapefile_name = os.path.splitext(shapefile.filename)[0]

for fName in os.listdir(dst_dir):
    zip.write(os.path.join(dst_dir, fName), fName)

zip.close()
```

Отметим, что, для того чтобы сохранить содержимое архива ZIP, мы используем временный файл, на который ссылается переменная temp. Мы вернем содержимое временного файла в веб-браузер пользователя, как только процесс экспорта закончится.

Удаление временных файлов

Далее нам нужно после себя подчистить, удалив файл фигур, который мы создали ранее:

```
shutil.rmtree(dst_dir)
```

Отметим, что нам не нужно удалять временный архив ZIP, поскольку он удаляется автоматически модулем `tempfile`, когда файл закрывается.

Возвращение архива zip пользователю

На последнем этапе экспорта файла фигур нужно отправить архив ZIP в веб-браузер пользователя, чтобы обеспечить возможность его скачать на компьютер пользователя. Для этого мы создадим специальный тип объекта `HttResponse` под названием `FileResponse`, который используется для скачивания файлов. Сначала нам нужно приготовить временный файл, которым объект `FileResponse` мог бы воспользоваться:

```
temp.flush()
temp.seek(0)
```

Благодаря этому все содержимое файла будет записано на диск, и текущая позиция в файле будет установлена в его начало. В результате содержимое всего файла будет скачано.

Теперь можно подготовить объект `FileResponse`, чтобы веб-браузер пользователя скачал его:

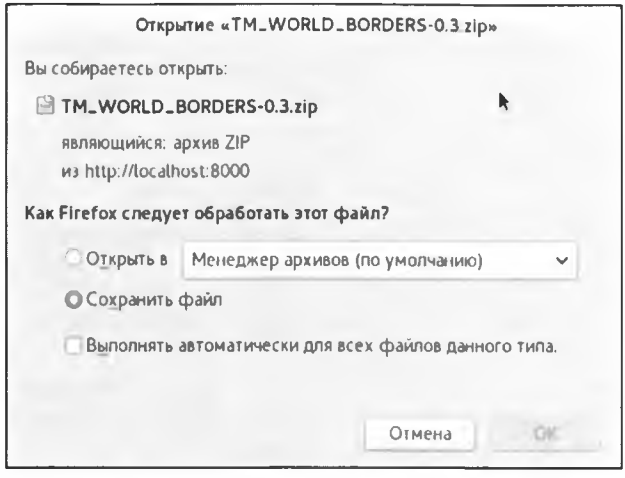
```
response = FileResponse(temp)
response['Content-type'] = "application/zip"
response['Content-Disposition'] = \
    "attachment; filename=" + shapefile_name + ".zip"
return response
```

Как видите, мы задаем значения заголовка HTTP-отклика, который обозначает, что мы возвращаем файловое вложение. Он заставляет браузер пользователя скачать файл, не пытаясь вывести его на экран. Кроме того, в качестве имени скачиваемого файла мы используем имя исходного файла фигур.

На этом определение функции `export_data()` завершено. Осталось сделать всего одну вещь: добавить в начало модуля `shapefile10.py` следующий ниже оператор импорта:

```
from django.http import FileResponse
```

Мы, наконец, закончили реализацию опции экспорта файла фигур. Проверьте ее, запустив сервер и нажав на гиперссылку **Экспорт** напротив одного из ваших файлов фигур. Если все выполнено правильно, то через небольшую паузу вам будет предложено сохранить на диск архив ZIP вашего файла фигур:



Заключение

В этой главе мы продолжили процесс реализации системы ShapeEditor, добавив три важные составляющие: режим просмотра данных в виде списка и функционал импорта и экспорта файлов фигур. Хотя они не очень-то интересные, их роль в системе ShapeEditor ключевая.

Во время реализации этих функций мы научились использовать язык шаблонной обработки веб-платформы Django для вывода списка записей внутри веб-страницы. Мы узнали, как использовать модуль `zipfile` стандартной библиотеки для извлечения содержимого выгруженного на сервер файла фигур, перед тем как его открыть при помощи библиотеки `OGR`, и мы обсудили понятие завертывания и развертывания геометрий, чтобы справиться с причудливым способом, которым они хранятся в формате файла фигур. Наконец, мы научились использовать библиотеку `OGR` для создания нового файла фигур, который можно сжать при помощи библиотеки `zipfile` перед его возвратом вызывающей стороне на основе веб-интерфейса Django.

Решив все вопросы, связанные с этой функциональностью, мы теперь можем обратиться к самым интересным компонентам системы ShapeEditor: программному коду, который выводит геометрии на экран компьютера и позволяет пользователю их редактировать, используя для этого интерфейс скользящей карты. Эта тема будет в центре нашего внимания в последней главе данной книги.

Глава 13

ShapeEditor – выбор и правка геообъектов

В этой последней главе мы займемся реализацией оставшегося функционала системы ShapeEditor. Значительная часть этой главы посвящена использованию библиотеки OpenLayers и созданию сервера сборных цифровых карт, которые позволяют визуализировать карту вместе со всеми геообъектами файла фигур и предоставляют пользователю возможность выбирать геообъект, нажимая на нем кнопкой мыши. Мы также займемся реализацией функционала добавления, правки и удаления геообъектов, а завершим главу рассмотрением приемов использования системы ShapeEditor для работы с геоданными и анализом того, каким образом она может послужить трамплином для ваших собственных инициатив в области разработки геоприложений.

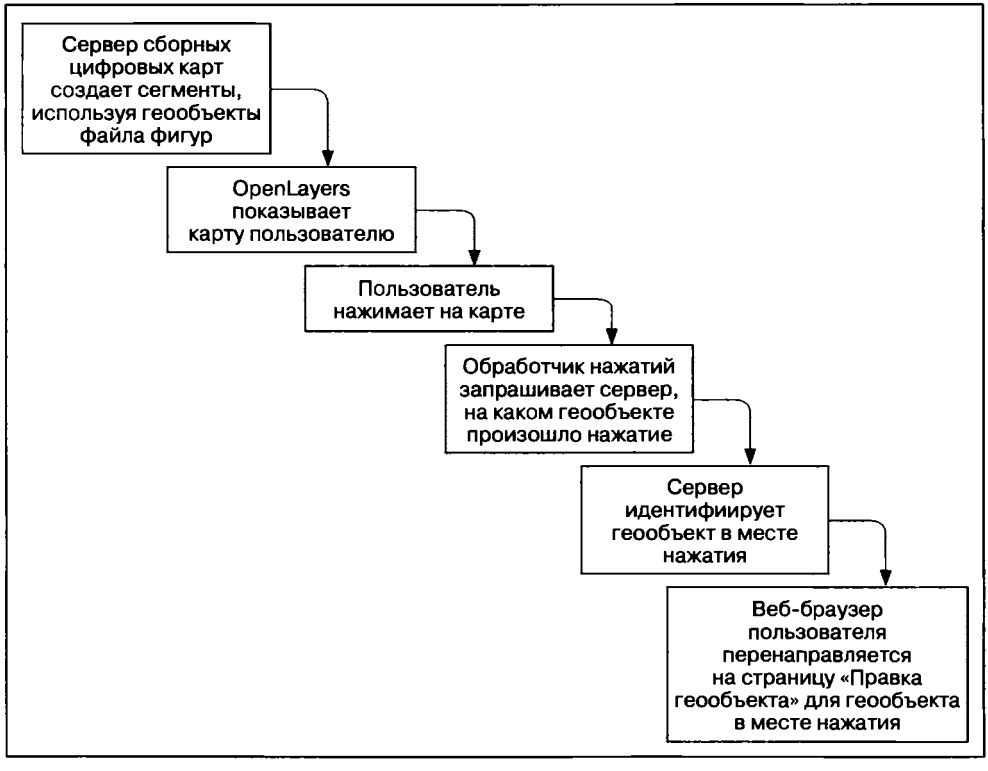
В этой главе мы узнаем, как:

- реализовать сервер сборных цифровых карт при помощи библиотеки Mapnik и модуля GeoDjango;
- использовать библиотеку OpenLayers, чтобы выводить на веб-страницу скользящую карту;
- написать собственный обработчик нажатий для библиотеки OpenLayers;
- использовать технологию AJAX для отправки запросов на сервер;
- выполнять пространственные запросы при помощи модуля GeoDjango;
- использовать в своем собственном приложении встроенные в GeoDjango виджеты редактирования;
- редактировать геоданные при помощи встроенных в GeoDjango виджетов редактирования;
- настраивать интерфейс для виджетов редактирования GeoDjango;
- добавлять и удалять записи в веб-приложении Django.

Начнем реализацию с программного кода, который позволяет пользователю выбирать геообъект для правки.

Выбор геообъекта для правки

Как уже обсуждалось в разделе «*Разработка системы ShapeEditor*» *главы 11* «*Собираем все вместе – полнофункциональная картографическая система*», встроенные картографические виджеты расширения GeoDjango могут за один раз выводить на экран всего один геообъект. Чтобы вывести на экран карту со всеми геообъектами файла фигур сразу, нам нужно наряду с сервером сборных цифровых карт и собственным AJAX-ориентированным обработчиком нажатий напрямую использовать библиотеку OpenLayers. В своей основе последовательность операций будет сводиться к следующему:



Начнем с реализации сервера сборных цифровых карт, а затем проанализируем, с чем сопряжено использование библиотеки OpenLayers, а также рассмотрим реализацию собственного обработчика нажатий и нескольких строк программного кода AJAX на стороне сервера для ответа на нажатия пользователя на карте.

Реализация сервера сборных цифровых карт

Как уже обсуждалось в *главе 10* «*Инструменты для разработки геопространственных веб-приложений*», протокол **службы сборных цифровых карт**, англ. тер-

мин Tile Map Service (**TMS**), – это просто протокол веб-службы на основе архитектуры RESTful, предназначенный для раздачи сборных цифровых карт. Протокол TMS содержит вызовы, которые отождествляют различные карты, которые могут быть визуализированы, и информацию об имеющихся сегментах цифровой карты, а также обеспечивает доступ непосредственно к самим изображениям сегментов.

Давайте кратко рассмотрим терминологию, используемую протоколом TMS.

- **Сервер сборных цифровых карт** (tile map server) – универсальный веб-сервер, в котором реализован протокол TMS.
- **Служба сборных цифровых карт** (tile map service) обеспечивает доступ к конкретному подмножеству карт. На одном сервере сборных цифровых карт может быть размещено несколько служб сборных цифровых карт.
- **Сборная карта** (tile map) – полная карта всей поверхности Земли или ее части, показывающая конкретное подмножество геообъектов или стилизованная определенным образом. Служба сборных цифровых карт может обеспечивать доступ к нескольким сборным цифровым картам.
- **Лист сборной карты** (tileset) – коллекция (подмножество) сегментов, показывающая конкретную сборную карту на конкретном масштабном уровне (уровне приближения).
- **Сегмент сборной карты** (tile) – единичный фрагмент изображения карты, представляющий небольшой участок карты в составе листа сборной карты.

Хотя при первом приближении эта терминология может показаться запутанной, но на самом деле все не так уж и сложно. Мы займемся реализацией сервера сборных цифровых карт всего с одной службой сборных цифровых карт, которую мы назовем веб-службой сборных карт ShapeEditor. Для каждого выгруженного на сервер файла фигур будет всего одна сборная карта, и мы будем поддерживать листы сборной карты для стандартного диапазона масштабных уровней. Наконец, мы воспользуемся библиотекой Mapnik для визуализации отдельных сегментов внутри листа сборной карты.

Следуя принятому в веб-платформе Django принципу разбивки большой и сложной системы на отдельные автономные приложения, мы создадим сервер сборных цифровых карт в рамках проекта `shapeEditor` как отдельное приложение. Начнем с того, что при помощи команды `cd` поменяем текущее месторасположение на каталог проекта `shapeEditor` и затем наберем следующее:

```
python3 manage.py startapp tms
```

Эта команда создаст приложение `tms` в каталоге верхнего уровня как приложение многократного использования. Переместите только что созданный каталог в подкаталог `shapeEditor`, набрав следующую ниже команду:

```
mv tms shapeEditor
```

Она сделает `tms` приложением, ограниченным только нашим проектом. Как обычно, нам не понадобятся модули `admin.py` и `tests.py` из каталога `tms`, поэтому просто удалите их.

Далее нам нужно это приложение активировать. Откройте модуль settings.py нашего проекта в текстовом редакторе и добавьте в конец списка INSTALLED_APPS следующую ниже запись:

```
'shapeEditor.tms',
```

Далее нам понадобится сделать URL-адреса нашего сервера сборных цифровых карт доступными в качестве составной части проекта shapeEditor. Для этого в текстовом редакторе откройте модуль верхнего уровня urls.py (расположенный в главном каталоге shapeEditor) и добавьте в список urlpatterns следующую ниже запись:

```
url(r'^tms/', include(shapeEditor.tms.urls)),
```

Вам также понадобится добавить в начало этого модуля следующий ниже оператор импорта:

```
import shapeEditor.tms.urls
```

Теперь нам нужно определить отдельные URL-адреса, предоставляемые нашим приложением сервера сборных цифровых карт. Для этого в каталоге tms создайте новый модуль с именем urls.py и наберите туда следующий фрагмент кода:

```
# URLConf for the shapeEditor.tms application.

from django.conf.urls import url
from shapeEditor.tms.views import *

urlpatterns = [
    url(r'^$',
        root), # "/tms" вызывает root()
    url(r'^(?P<version>[0-9.]+)$',
        service), # например, "/tms/1.0" вызывает service(version="1.0")
    url(r'^(?P<version>[0-9.]+)/(?P<shapefile_id>\d+)$',
        tileMap), # например, "/tms/1.0/2" вызывает
                  # tileMap(version="1.0", shapefile_id=2)
  url(r'^(?P<version>[0-9.]+)/' +
      r'(?P<shapefile_id>\d+)/(?P<zoom>\d+)/' +
      r'(?P<x>\d+)/(?P<y>\d+)\.png$',
      tile), # например, "/tms/1.0/2/3/4/5" вызывает
             # tile(version="1.0", shapefile_id=2, zoom=3, x=4, y=5)
]
```

Эти URL-шаблоны более сложные, чем те, что мы использовали ранее, потому что теперь из URL-адреса мы извлекаем параметры. Например, рассмотрим следующий ниже URL-адрес:

```
http://127.0.0.1:8000/tms/1.0
```

Вследствие характера настройки наших URL-модулей первая часть этого URL-адреса (http://i27.0.0.i:8000/tms/) сообщает веб-платформе Django, что этот

URL-адрес будет обработан модулем tms.urls. Остальная часть URL, 1.0, совпадет со вторым регулярным выражением модуля urls.py:

```
^(?P<version>[0-9.]+)$
```

Приведенное выше регулярное выражение извлечет участок URL-адреса с 1.0 и присвоит его параметру version, который затем будет передан просмотровой функции, связанной с этим URL-шаблоном, следующим образом:

```
service(version="1.0")
```

Благодаря этому каждый URL-шаблон в рамках нашего приложения tms ставит поступающему URL-адресу с архитектурой RESTful в соответствие просмотровую функцию. Содержащиеся в программном коде комментарии показывают примеры того, как регулярные выражения отображаются на просмотровые функции.

Теперь настроим эти просмотровые функции. Откройте в текстовом редакторе модуль views.py из каталога tms и добавьте в этот модуль следующий ниже фрагмент кода:

```
from django.http import HttpResponse

def root(request):
    return HttpResponse("Сервер сборных цифровых карт")

def service(request, version):
    return HttpResponse("Служба сборных цифровых карт")

def tileMap(request, version, shapefile_id):
    return HttpResponse("Сборная карта")

def tile(request, version, shapefile_id, zoom, x, y):
    return HttpResponse("Сегмент сборной карты")
```

Безусловно, это всего лишь заготовки просмотровых функций, но они обеспечивают нам базовую структуру нашего сервера сборных цифровых карт.

Чтобы проверить, что он работает, запустите сервер ShapeEditor, выполнив команду python3 manage.py runserver, и направьте ваш веб-браузер по URL-адресу http://127.0.0.1:8000/tms. Вы должны увидеть текст, который вы ввели в свою заготовку просмотровой функции root().

Заставим эту просмотровую функцию верхнего уровня сделать что-то полезное. Вернитесь к модулю views.py приложения tms и измените функцию root() так, чтобы она выглядела следующим ниже образом:

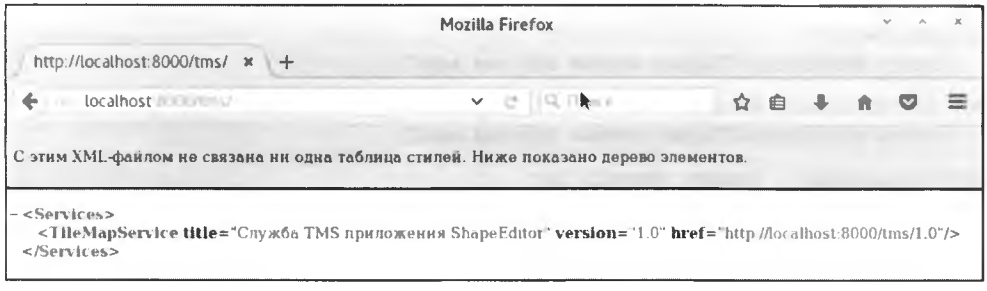
```
def root(request):
    try:
        baseURL = request.build_absolute_uri()
        xml = []
        xml.append('<?xml version="1.0" encoding="utf-8" ?>')
        xml.append('<Services>')
        xml.append(' <TileMapService ' +
```

```
                'title="Служба TMS приложения ShapeEditor" ' +
                'version="1.0" href="' + baseURL + '/1.0"/>')
        xml.append('</Services>')
        response = "\n".join(xml)
        return HttpResponse(response, content_type="text/xml")
    except:
        traceback.print_exc()
    return HttpResponse("Ошибка")
```

Вам также придется добавить в начало модуля следующий ниже оператор импорта:

```
import traceback
```

Эта просмотровая функция возвращает отклик сервера в формате XML, описывающий одну-единственную службу сборных цифровых карт, поддерживаемую нашим сервером TMS. Служба сборных цифровых карт определяется по номеру версии, `1.0` (любая служба сборных цифровых карт, как правило, идентифицируются по номеру версии). Если вы теперь пройдете на `http://127.0.0.1:8000/tms`, то увидите в вашем веб-браузере отклик сервера TMS:



На снимке (внутри XML-элемента Services) показан список всех служб сборных цифровых карт, которые предлагает этот сервер TMS. Библиотека OpenLayers будет использовать его для доступа к нашей службе сборных цифровых карт.

Обработка ошибок

Отметим, что мы завернули нашу просмотровую функцию TMS в оператор `try...except` и использовали модуль `traceback` стандартной библиотеки, чтобы распечатать исключение, если что-нибудь пойдет не так. Мы делаем это, потому что наш программный код вызывается непосредственно из библиотеки OpenLayers, используя для этого технологию AJAX; веб-платформа Django с готовностью обрабатывает исключения и возвращает вызывающей стороне HTML-страницу ошибки, но в данном случае, если в вашем программном коде будет ошибка, библиотека OpenLayers не покажет эту страницу. Напротив, вместо карты вы увидите значки отсутствующих изображений, а сама ошибка останется тайной за семью печатями.

Завертывая наш програмный код на Python в оператор `try...except`, мы можем отлавливать в программе любые исключения и их распечатывать. В результате

это приведет к тому, что ошибка появится в журнале веб-сервера Django, и мы сможем увидеть, что пошло не так. Этот полезный прием стоит использовать всегда, когда вы пишете обработчики запросов AJAX на Python.

Теперь мы готовы реализовать непосредственно саму службу сборных цифровых карт. Откройте модуль `view.py` еще раз и измените функцию `service()` так, чтобы она выглядела следующим ниже образом:

```
def service(request, version):
    try:
        if version != "1.0":
            raise Http404

        baseURL = request.build_absolute_uri()
        xml = []
        xml.append('<?xml version="1.0" encoding="utf-8" ?>')
        xml.append('<TileMapService version="1.0" services="' +
                   baseURL + '">')
        xml.append(' <Title>Служба TMS приложения ShapeEditor' +
                   '</Title>')
        xml.append(' <Abstract></Abstract>')
        xml.append(' <TileMaps>')
        for shapefile in Shapefile.objects.all():
            id = str(shapefile.id)
            xml.append(' <TileMap title="' +
                       shapefile.filename + '"')
            xml.append('              srs="EPSG:4326"')
            xml.append('              href="'+baseURL+'/'+id+'"/>')
        xml.append(' </TileMaps>')
        xml.append('</TileMapService>')
        response = "\n".join(xml)
        return HttpResponse(response, content_type="text/xml")
    except:
        traceback.print_exc()
        return HttpResponse("Ошибка")
```

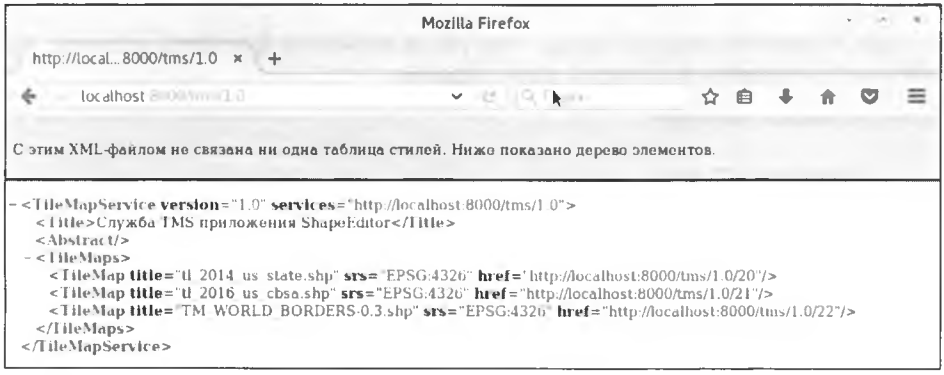
Вам также придется добавить в начало модуля следующие ниже операторы импорта:

```
from django.http import Http404
from shapeEditor.shared.models import Shapefile
```

Отметим, что эта функция поднимает исключение `Http404`, если номер версии неправильный. Это исключение поручает веб-платформе Django вернуть ошибку HTTP «Страница не найдена», которая является стандартным ответом на ошибку, когда был использован неправильный URL-адрес.

При условии, что номер версии корректен, мы итеративно просматриваем различные объекты `Shapefile` в базе данных, выводя список всех выгруженных на сервер файлов фигур в виде XML-элементов для сборных карт TileMap.

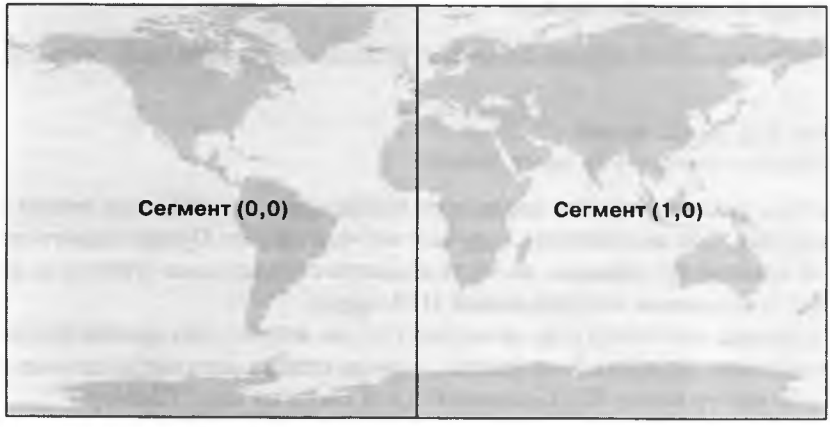
Если сохранить этот файл и ввести в веб-браузере `http://127.0.0.1:8000/tms/1.0`, то в составе XML-элемента TileMaps вы должны увидеть список всех имеющихся сборных карт:



Далее мы должны реализовать функцию `tileMap()`, чтобы вывести на экран листы заданной сборной цифровой карты. Правда, прежде чем мы сможем это сделать, нам придется немного разобраться в понятии **масштабных уровней**, или уровней приближения.

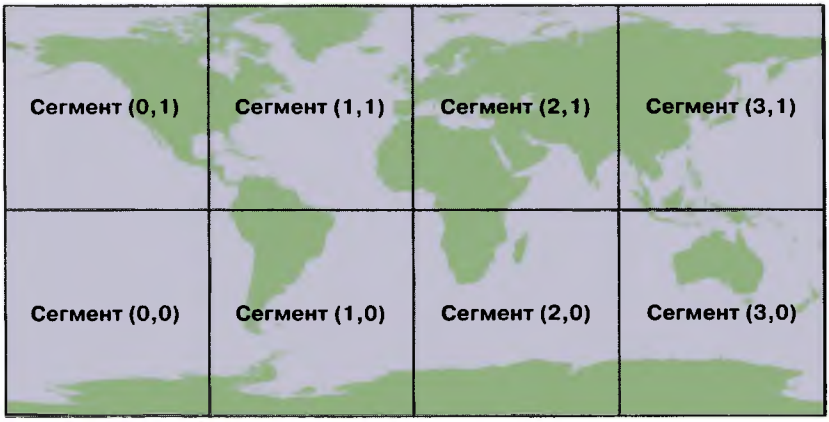
Как мы уже убедились, скользящая карта позволяет пользователю увеличивать и уменьшать масштаб просмотра содержимого карты. Масштабирование выполняется путем управления уровнем приближения карты. Как правило, масштабный уровень определяется как простое число: уровень приближения, равный 0, обозначает, что карта максимально уменьшена, уровень приближения 1 – что карта увеличена на единицу, и т. д.

Начнем с того, что рассмотрим максимально уменьшенную карту (то есть с масштабным уровнем 0). В этом случае, чтобы охватить всю поверхность Земли, нам потребуются всего два сегмента сборной карты:



Каждый сегмент на этом масштабном уровне будет охватывать 180° широты и долготы. Если бы каждый сегмент был квадратом со стороной 256 пикселов, то это означало бы, что каждый пиксел покрывал бы 180 / 256 = 0.703125 единицы измерения карты, где в данном случае единица измерения карты – это градус широты или долготы. Это число будет иметь крайне важное значение, когда дело дойдет до вычисления сборных карт.

Далее при каждом увеличении масштаба (например, при переходе с масштабного уровня 0 к масштабному уровню 1) ширина и высота видимой области делятся пополам. Например, на масштабном уровне 1 поверхность Земли была бы выведена как следующая серия из восьми сегментов:



Следуя данной схеме, мы можем вычислить число единиц измерения карты, которые охватывают один пиксел карты для заданного масштабного уровня, используя для этого нижеследующую формулу:

$$\text{Количество единиц измерения карты на пиксел} = \frac{0.703125}{2^{\text{масштабный уровень}}}.$$

С учетом того, что мы будем использовать эту формулу в нашем сервере TMS, пойдем дальше и добавим в конец нашего модуля `tms/views.py` следующий ниже программный код:

```
def _unitsPerPixel(zoomLevel):
    return 0.703125 / math.pow(2, zoomLevel)
```

Отметим, что имя функции начинается с символа подчеркивания; это стандартное для Python условное обозначение, используемое для именования частных (закрытых) функций внутри модуля.

Вам также придется добавить в начало файла оператор импорта `import math`.

Далее нам нужно добавить в начало модуля несколько констант, чтобы задать размер каждого сегмента и количество поддерживаемых масштабных уровней:

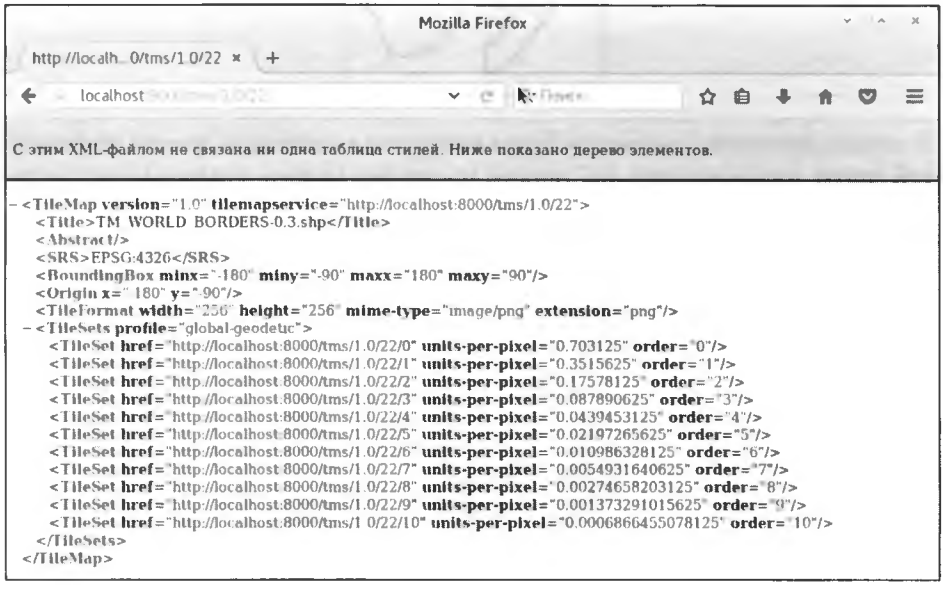
```
MAX_ZOOM_LEVEL = 10
TILE_WIDTH     = 256
TILE_HEIGHT    = 256
```

С учетом всего этого мы, наконец, готовы реализовать функцию `tileMap()`, которая для заданного файла фигур будет возвращать информацию об имеющихся листах сборной карты. Отредактируйте эту функцию, чтобы она выглядела следующим ниже образом:

```
def tileMap(request, version, shapefile_id):
    if version != "1.0":
        raise Http404

    try:
        shapefile = Shapefile.objects.get(id=shapefile_id)
    except Shapefile.DoesNotExist:
        raise Http404

    try:
        baseURL = request.build_absolute_uri()
        xml = []
        xml.append('<?xml version="1.0" encoding="utf-8" ?>')
        xml.append('<TileMap version="1.0" ' +
                   'tilemapservice="' + baseURL + '">')
        xml.append(' <Title>' + shapefile.filename + '</Title>')
        xml.append(' <Abstract></Abstract>')
        xml.append(' <SRS>EPSG:4326</SRS>')
        xml.append(' <BoundingBox minx="-180" miny="-90" ' +
                   'maxx="180" maxy="90"/>')
        xml.append(' <Origin x="-180" y="-90"/>')
        xml.append(' <TileFormat width="' + str(TILE_WIDTH) +
                   '" height="' + str(TILE_HEIGHT) + '" ' +
                   'mime-type="image/png" extension="png"/>')
        xml.append(' <TileSets profile="global-geodetic">')
        for zoomLevel in range(0, MAX_ZOOM_LEVEL+1):
            href = baseURL + "/{}".format(zoomLevel)
            unitsPerPixel = "{}".format(_unitsPerPixel(zoomLevel))
            order = "{}".format(zoomLevel)
            xml.append(' <TileSet href="' + href + '" ' +
                       'units-per-pixel="'+ unitsPerPixel + '"' +
                       ' order="' + order + '"/>')
        xml.append(' </TileSets>')
        xml.append('</TileMap>')
        response = "\n".join(xml)
        return HttpResponse(response, content_type="text/xml")
    except:
        traceback.print_exc()
        return HttpResponse("Ошибка")
```

Как видите, мы начинаем с выявления нескольких основных ошибок, касающихся версии сервера и идентификатора файла фигур, и затем в итеративном ре-

жиме просматриваем доступные масштабные уровни, чтобы предоставить информацию об имеющихся листах сборной карты. Если вы сохраните внесенные вами изменения и в вашем веб-браузере введете `http://i27.0.0.i:8000/tms/1.0/22`, то вы должны увидеть следующую ниже информацию о сборной карте для файла фигур с идентификатором записи 22:



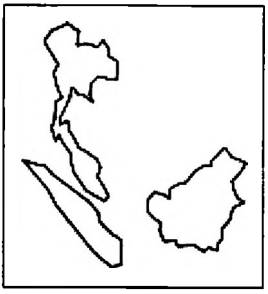
Отметим, что мы предоставляем в общей сложности одиннадцать масштабных уровней, от 0 до 10, с соответствующим образом рассчитанными единицами измерения карты на пиксел для каждого масштабного уровня.

Мы только что реализовали три из четырех просмотровых функций, требующихся для реализации нашего собственного сервера сборных карт. Для заключительной функции `tile()` мы напишем наш собственный инструмент визуализации сегментов. Функция `tile()` получает версию службы сборных карт, идентификатор файла фигур, масштабный уровень и прямоугольные координаты x и y нужного сегмента:

```
def tile(request, version, shapefile_id, zoom, x, y):
    ...
```

Эта функция должна сгенерировать соответствующий сегмент и вернуть результат визуализации вызывающей стороне. Прежде чем мы эту функцию реализуем, сделаем шаг назад и подумаем о том, как должна выглядеть визуализация цифровой карты.

Мы хотим, чтобы карта содержала контур различных геообъектов внутри заданного файла фигур. Однако эти геообъекты сами по себе не будут нести в себе какой-то особый смысл:



И только когда эти геообъекты будут выведены в контексте, то есть на фоне **базовой карты**, или подложки, мы увидим то, что они по идее должны представлять:

По этой причине нам придется отображать базовую карту, на которую будут выводиться непосредственно сами геообъекты. Создадим эту базовую карту и затем будем ее использовать для визуализации сборной карты вместе с геообъектами файла фигур.

Настройка базовой карты

Для нашей базовой карты мы будем пользоваться набором данных границ стран мира, который мы уже несколько раз использовали в этой книге. Этот набор данных при прямом увеличении масштаба выглядит не особо, вместе с тем

он хорошо подходит в качестве базовой карты, на которую можно наносить геообъекты файла фигур.

Мы начнем с создания модели базы данных, которая будет содержать данные базовой карты. Поскольку базовая карта будет ограничена нашим приложением сервера сборных цифровых карт, нам нужно добавить ограниченную этим приложением таблицу базы данных. Для этого откройте в текстовом редакторе модуль `models.py` в каталоге приложения `tms` и измените этот файл так, чтобы он выглядел следующим ниже образом:

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

class BaseMap(models.Model):
    name = models.CharField(max_length=50)
    geometry = models.MultiPolygonField(srid=4326)

    objects = models.GeoManager()

    def __str__(self):
        return self.name
```

Не забудьте поменять оператор импорта `import` в начале файла.

Как видите, мы сохраняем названия стран и их геометрии, которые представлены составными многоугольниками MultiPolygon. Далее в окне командной строки при помощи команды `cd` перейдите в каталог вашего проекта и наберите следующие ниже команды:

```
% python3 manage.py makemigrations tms
% python3 manage.py migrate tms
```

В результате будет создана таблица базы данных, используемая объектом базовой карты `BaseMap`.

Располагая местом, где хранится базовая карта, теперь выполним импорт данных. Разместите копию файла фигур набора данных границ стран мира в какомнибудь удобном для вас месте и в открытом ранее окне командной строки наберите следующую ниже строку:

```
% python3 manage.py shell
```

В результате будет запущена интерактивная оболочка Python с настройками вашего проекта и путей доступа. Теперь создайте следующую ниже переменную, заменив текст абсолютным путем к файлу фигур с набором данных границ:

```
>>> shapefile = "/путь/к/TM_WORLD_BORDERS-0.3.shp"
```

Затем наберите следующее:

```
>>> from django.contrib.gis.utils import LayerMapping
>>> from shapeEditor.tms.models import BaseMap
>>> mapping = LayerMapping(BaseMap, shapefile, {'name' : "NAME",
'geometry' : "MULTIPOLYGON"}, transform=False, encoding="iso-8859-1")
>>> mapping.save(strict=True, verbose=True)
```

Мы используем модуль LayerMapping модуля GeoDjango, чтобы импортировать данные из этого файла фигур в нашу базу данных. Во время импортирования, которое займет несколько секунд, на экране будут показаны названия различных стран.

По завершении импортированные данные можно проверить, набирая команды в интерактивной оболочке. Например:

```
>>> print(BaseMap.objects.count())
246
>>> print(BaseMap.objects.all())
[<BaseMap: Antigua and Barbuda>, <BaseMap: Algeria>, <BaseMap:
Azerbaijan>, <BaseMap: Albania>, <BaseMap: Armenia>, <BaseMap: Angola>,
<BaseMap: American Samoa>, <BaseMap: Argentina>, <BaseMap: Australia>,
<BaseMap: Bahrain>, <BaseMap: Barbados>, <BaseMap: Bermuda>, <BaseMap:
Bahamas>, <BaseMap: Bangladesh>, <BaseMap: Belize>, <BaseMap: Bosnia and
Herzegovina>, <BaseMap: Bolivia>, <BaseMap: Burma>, <BaseMap: Benin>,
<BaseMap: Solomon Islands>, '...(remaining elements truncated)...']
```

Не стесняйтесь еще немного поэкспериментировать, если хотите; учебное руководство по веб-платформе Django содержит несколько примеров исследования ваших объектов данных при помощи интерактивной оболочки. Когда вы закончите, нажмите **Ctrl+D**, чтобы выйти.

Поскольку эта базовая карта будет составной частью непосредственно самого проекта ShapeEditor (приложение не будет выполняться без него), было бы хорошо, если бы веб-платформа Django могла рассматривать эти данные как составную часть исходного кода проекта. В этом смысле, если бы нам когда-нибудь пришлось восстанавливать базу данных с нуля, базовая карта была бы переустановлена автоматически.

Веб-платформа Django позволяет вам это делать, создавая так называемые **оснастки** (fixtures), или данные предварительной настройки. Оснастка – это набор данных, который может быть загружен в базу данных по требованию, вручную либо автоматически, при инициализации базы данных. Мы сохраним наши данные базовой карты в оснастке, с тем чтобы в случае необходимости веб-платформа Django могла их перезагрузить.

Создайте внутри каталога приложения tms каталог fixtures. Затем в окне терминала при помощи команды cd перейдите в каталог проекта shapeEditor и наберите следующее:

```
% python3 manage.py dumpdata tms > shapeEditor/tms/fixtures/initial_data.json
```

В результате для приложения tms будет создана оснастка с именем initial_data.json. Как следует из ее названия – исходные данные, содержимое этой оснастки будет загружено автоматически, если веб-платформе Django придется когда-либо инициализировать базу данных повторно.

Располагая базовой картой, теперь ей можно воспользоваться для реализации нашего программного кода визуализации сегментов.

Визуализация сегментов

Используя наши познания библиотеки Mapnik, приступим к реализации функции `tile()` сервера TMS. Наша сгенерированная карта будет состоять из двух слоев: **базового слоя** с базовой картой (подложкой) и **слоя геообъектов** с геообъектами из импортированного файла фигур. Учитывая, что все наши данные хранятся в базе данных PostGIS, для обоих слоев мы будем использовать источник данных `mapnik.PostGIS`.

Наша функция `tile()` состоит из пяти этапов.

1. Разбор параметров запроса.
2. Настройка карты.
3. Задание базового слоя.
4. Задание слоя геообъектов.
5. Визуализация карты.

Разберем по очереди каждый из них.

Разбор параметров запроса

Откройте модуль `views.py` приложения `tms` в текстовом редакторе и удалите из функции `tile()` программный код заготовки. Мы добавим наш программный код разбора параметров запроса шаг за шагом, начиная с элементарной проверки, которая обеспечит проверку корректности номера версии и существования файла фигур, и снова завернем наш программный код в оператор `try...except` для отлова опечаток и других ошибок:

```
try:
    if version != "1.0":
        raise Http404

    try:
        shapefile = Shapefile.objects.get(id=shapefile_id)
    except Shapefile.DoesNotExist:
        raise Http404
```

Теперь нам нужно конвертировать параметры запроса (которые веб-платформа Django передает нам как строковые значения) в целые числа, с тем чтобы мы могли с ними работать:

```
zoom = int(zoom)
x = int(x)
y = int(y)
```

Теперь можно проверить правильность масштабного уровня:

```
if zoom < 0 or zoom > MAX_ZOOM_LEVEL:
    raise Http404
```

На следующем этапе нам нужно конвертировать предоставленные параметры `x` и `y` в значения минимальной и максимальной широты и долготы, покрываемые сегментом. Для этого нам необходимо воспользоваться определенной ранее функ-

цией _unitsPerPixel(), которая вычисляет величину поверхности Земли, охватываемую сегментом для текущего масштабного уровня:

```
xExtent = _unitsPerPixel(zoom) * TILE_WIDTH
yExtent = _unitsPerPixel(zoom) * TILE_HEIGHT

minLong = x * xExtent - 180.0
minLat = y * yExtent - 90.0
maxLong = minLong + xExtent
maxLat = minLat + yExtent
```

Наконец, можно добавить элементарную проверку ошибок, которая обеспечит проверку допустимости координаты сегмента:

```
if (minLong < -180 or maxLong > 180 or
    minLat < -90 or maxLat > 90):
    raise Http404
```

Настройка карты

Теперь мы готовы создать объект mapnik.Map, который будет представлять карту. Это реализуется достаточно просто:

```
map = mapnik.Map(TILE_WIDTH, TILE_HEIGHT,
                 "+proj=longlat +datum=WGS84")
map.background = mapnik.Color("#7391ad")
```

Задание базового слоя

Теперь нам нужно определить слой, в котором визуализируется наша базовая карта. Для этого нам следует настроить для слоя его источник данных mapnik.PostGIS:

```
dbSettings = settings.DATABASES['default']

datasource = \
  mapnik.PostGIS(user=dbSettings['USER'],
                 password=dbSettings['PASSWORD'],
                 dbname=dbSettings['NAME'],
                 table='tms_basemap',
                 srid=4326,
                 geometry_field="geometry",
                 geometry_table='tms_basemap')
```

Как видите, мы получаем имя базы данных, имя пользователя и пароль из модуля настроек settings нашего проекта. И затем, используя эти настройки, создаем источник данных PostGIS. Имея источник данных, теперь мы можем создать непосредственно сам базовый слой:

```
baseLayer = mapnik.Layer("baseLayer")
baseLayer.datasource = datasource
baseLayer.styles.append("baseLayerStyle")
```

Теперь нам нужно настроить стиль слоя. В данном случае мы воспользуемся одним правилом с двумя символизаторами: символизатором многоугольников PolygonSymbolizer и символизатором линий LineSymbolizer, первый заполняет внутреннюю часть многоугольников базовой карты, второй выводит контуры многоугольников:

```
rule = mapnik.Rule()

rule.symbols.append(
    mapnik.PolygonSymbolizer(mapnik.Color("#b5d19c")))
rule.symbols.append(
    mapnik.LineSymbolizer(mapnik.Color("#404040"), 0.2))

style = mapnik.Style()
style.rules.append(rule)
```

Наконец, можно добавить в карту базовый слой и его стиль:

```
map.append_style("baseLayerStyle", style)
map.layers.append(baseLayer)
```

Задание слоя геообъектов

Наша следующая задача состоит в том, чтобы добавить еще один слой для нанесения геообъектов файла фигур на карту. Для нового слоя еще раз настроим источник данных `mapnik.PostGIS`:

```
geometry_field = \
  utils.calc_geometry_field(shapefile.geom_type)
  query = '(select ' + geometry_field \
        + ' from "shared_feature" where' \
        + ' shapefile_id=' + str(shapefile.id) + ') as geom'

datasource = \
  mapnik.PostGIS(user=dbSettings['USER'],
                 password=dbSettings['PASSWORD'],
                 dbname=dbSettings['NAME'],
                 table=query,
                 srid=4326,
                 geometry_field=geometry_field,
                 geometry_table='shared_feature')
```

В данном случае мы вызываем функцию `utils.calc_geometry_field()`, которая извлекает из таблицы `shared_feature` поле с геометрией, которую мы собираемся показать.

Теперь мы готовы создать непосредственно сам новый слой:

```
featureLayer = mapnik.Layer("featureLayer")
featureLayer.datasource = datasource
featureLayer.styles.append("featureLayerStyle")
```

Далее нам нужно задать стили для слоя геообъектов. Как и прежде, у нас будет всего одно правило, но в данном случае мы воспользуемся разными символизаторами в зависимости от типа выводимого геообъекта:

```
rule = mapnik.Rule()

if shapefile.geom_type in ["Point", "MultiPoint"]:
    rule.symbols.append(mapnik.PointSymbolizer())
elif shapefile.geom_type in ["LineString", "MultiLineString"]:
    rule.symbols.append(
        mapnik.LineSymbolizer(mapnik.Color("#000000"), 0.5))
elif shapefile.geom_type in ["Polygon", "MultiPolygon"]:
    rule.symbols.append(
        mapnik.PolygonSymbolizer(mapnik.Color("#f7edee")))
    rule.symbols.append(
        mapnik.LineSymbolizer(mapnik.Color("#000000"), 0.5))

style = mapnik.Style()
style.rules.append(rule)
```

Наконец, мы можем добавить в карту наш новый слой геообъектов:

```
map.append_style("featureLayerStyle", style)
map.layers.append(featureLayer)
```

Визуализация сегментов сборной карты

Мы проанализировали процесс формирования изображения цифровой карты при помощи библиотеки Mapnik в *главе 7 «Генерирование карт при помощи Python и библиотеки Mapnik»*. В своей основе процесс визуализации сегмента сборной карты аналогичный, за исключением того, что мы не будем сохранять результаты на диск в графический файл. Вместо этого мы создадим объект `mapnik.Image`, преобразуем его в исходные графические данные в формате PNG и вернем эти данные назад вызывающей стороне при помощи объекта `HttpResponse`:

```
map.zoom_to_box(mapnik.Box2d(minLong, minLat,
                             maxLong, maxLat))
image = mapnik.Image(TILE_WIDTH, TILE_HEIGHT)
mapnik.render(map, image)
imageData = image.tostring('png')

return HttpResponse(imageData, content_type="image/png")
```

Теперь осталось лишь добавить в конец функции программный код обнаружения ошибки:

```
except:
    traceback.print_exc()
    return HttpResponse("Ошибка")
```

На этом реализация функции `tile()` сервера сборных цифровых карт завершена. Теперь наведем порядок и проведем небольшое тестирование.

Завершение работы над сервером сборных цифровых карт

Поскольку в нашем модуле `views.py` мы обращались к нескольким новым модулям, нам следует в начало файла добавить несколько дополнительных операторов импорта:

```
import mapnik
from django.conf import settings
from shapeEditor.shared import utils
```

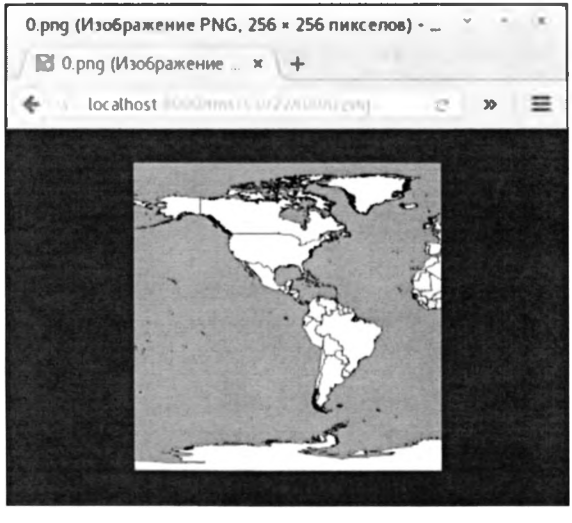
В принципе, наш сервер сборных цифровых карт теперь должен быть готовым к работе. Давайте его проверим. Если в данный момент веб-сервер Django не выполняется, то перейдите при помощи команды `cd` в каталог проекта `shapeEditor` и наберите следующую ниже команду:

```
% python3 manage.py runserver
```

Запустите веб-браузер и в строке поиска введите следующий ниже URL-адрес:

```
http://127.0.0.1:8000/tms/1.0/2/0/0/0.png
```

Если все выполнено правильно, то в веб-браузере вы должны увидеть сегмент сборной карты размером 256×256 пикселов:



Проблемы?

Если произошла ошибка, то есть две вероятные причины, которые ее вызвали: вы, возможно, сделали ошибку при наборе программного кода либо идентификатор файла фигур неправильный. Проверьте протокол работы веб-сервера в окне терминала, которое вы использовали для выполнения команды `python manage.py runserver`: когда в среде Python происходит исключение или ошибка при исполнении программы, то в этом окне распечатывается обратная трассировка,

которая сообщает вам о синтаксической или иной ошибке и было ли вызвано исключение `Http404`.

Если же вы получили исключение `Http404`, то, скорее всего, оно было вызвано неправильным идентификтором файла фигур. URL-адрес имеет следующую структуру:

```
http://путь/к/tms/<version>/<shapefile_id>/<zoom>/<x>/<y>.png
```

Если вы работали с главами книги по порядку, то идентификатор файла фигур с набором данных границ стран мира, который вы импортировали ранее, должен равняться 2, но если вы в промежутке импортировали другие файлы фигур либо, экспериментируя с веб-интерфейсом администратора, создали для файлов фигур другие записи, то у вас, скорее всего, будет другой идентификатор записи. Чтобы увидеть, какой идентификатор имеет заданный файл фигур, перейдите по `http://127.0.0.1:8000` и щелкните по гиперссылке **Правка** напротив нужного файла фигур. Вы увидите ошибку «**Страница не найдена**», но заключительная часть URL-адреса будет содержать идентификатор файла фигур. Замените идентификатор записи в предыдущем URL-адресе на корректный, и сегмент карты должен появиться.

Как только вы выйдете на этап, когда в своем веб-браузере увидите приведенное выше изображение, вас смело можно похлопать по плечу – поздравляем, вы только что реализовали свой собственный действующий сервер сборных цифровых карт!

Ниже приведена реализация рассмотренной выше функции `tile()` на основе декларативной парадигмы с использованием XML.

```
def tile(request, version, shapefile_id, zoom, x, y):
    try:
        # Разобрать и проверить параметры.
        if version != "1.0":
            raise Http404
        try:
            shapefile = Shapefile.objects.get(id=shapefile_id)
        except Shapefile.DoesNotExist:
            raise Http404

        zoom = int(zoom)
        x    = int(x)
        y    = int(y)

        if zoom < 0 or zoom > MAX_ZOOM_LEVEL: raise Http404

        xExtent = _unitsPerPixel(zoom) * TILE_WIDTH
        yExtent = _unitsPerPixel(zoom) * TILE_HEIGHT

        minLong = x * xExtent - 180.0
        minLat  = y * yExtent - 90.0
        maxLong = minLong + xExtent
        maxLat  = minLat  + yExtent

        if (minLong < -180 or maxLong > 180 or
```

```
            minLat < -90 or maxLat > 90):
            raise Http404

        # Задать базовый слой и слой геообъектов.
        map_string = '''<?xml version="1.0" encoding="utf-8"?>
<Map background-color="#7391ad" srs="+proj=longlat +datum=WGS84">
    <FontSet name="bold-fonts">
        <Font face-name="DejaVu Sans Bold" />
    </FontSet>
    <Style name="baseLayerStyle">
        <Rule>
            <PolygonSymbolizer fill="#b5d19c" />
            <LineSymbolizer stroke="#404040" stroke-width="0.2" />
        </Rule>
    </Style>
    <Style name="featureLayerStyle">
        <Rule>
        <!--(Symbolizers)-->
        </Rule>
    </Style>
    <Layer name="baseLayer">
        <StyleName>baseLayerStyle</StyleName>
        <Datasource>
            <Parameter name="type">postgis</Parameter>
            <Parameter name="user">(User)</Parameter>
            <Parameter name="password">(Password)</Parameter>
            <Parameter name="dbname">(Dbname)</Parameter>
            <Parameter name="table">tms_basemap</Parameter>
            <Parameter name="geometry_field">geometry</Parameter>
            <Parameter name="geometry_table">tms_basemap</Parameter>
            <Parameter name="srid">4326</Parameter>
        </Datasource>
    </Layer>
    <Layer name="featureLayer">
        <StyleName>featureLayerStyle</StyleName>
        <Datasource>
            <Parameter name="type">postgis</Parameter>
            <Parameter name="user">(User)</Parameter>
            <Parameter name="password">(Password)</Parameter>
            <Parameter name="dbname">(Dbname)</Parameter>
            <Parameter name="table">(Query)</Parameter>
            <Parameter name="geometry_field">(Geometry_field)</Parameter>
            <Parameter name="geometry_table">shared_feature</Parameter>
            <Parameter name="srid">4326</Parameter>
        </Datasource>
    </Layer>
</Map>
'''
        # Получить информацию о БД.
```

```
        dbSettings = settings.DATABASES['default']
    user = dbSettings['USER']
    passwd = dbSettings['PASSWORD']
    dbname = dbSettings['NAME']
    geometry_field = utils.calc_geometry_field(shapefile.geom_type)
        query = '(select ' + geometry_field \
              + ' from "shared_feature" where' \
              + ' shapefile_id=' + str(shapefile.id) + ') as geom'

    symbolizer = ""
        if shapefile.geom_type in ["Point", "MultiPoint"]:
            symbolizer = '<PointSymbolizer />'
        elif shapefile.geom_type in ["LineString", "MultiLineString"]:
            symbolizer = '<LineSymbolizer stroke="#000000" stroke-width="0.5" />'
        elif shapefile.geom_type in ["Polygon", "MultiPolygon"]:
            symbolizer = \
               '<PolygonSymbolizer fill="#f7edee" />' +
               '<LineSymbolizer stroke="#000000" stroke-width="0.5" />'

        # Выполнить подстановку
    map_string = map_string.replace("(User)", user)
    map_string = map_string.replace("(Password)", passwd)
    map_string = map_string.replace("(Dbname)", dbname)
    map_string = map_string.replace("(Geometry_field)", geometry_field)
    map_string = map_string.replace("(Query)", query)
    map_string = map_string.replace("<!--(Symbolizers)-->", symbolizer)

        # Настроить карту.
    gmap = mapnik.Map(TILE_WIDTH, TILE_HEIGHT)
    mapnik.load_map_from_string(gmap, map_string)

        # В заключение визуализировать сегмент.
    #gmap.zoom_to_box(mapnik.Envelope(minX, minY, maxX, maxY))
        gmap.zoom_to_box(mapnik.Box2d(minLong, minLat, maxLong, maxLat))
        image = mapnik.Image(TILE_WIDTH, TILE_HEIGHT)
        mapnik.render(gmap, image)
        imageData = image.tostring('png')
        return HttpResponse(imageData, content_type="image/png")
      except:
        traceback.print_exc()
        return HttpResponse("Ошибка")
```

Отображение карты при помощи библиотеки OpenLayers

Располагая готовым к работе сервером TMS, теперь можно воспользоваться библиотекой OpenLayers для вывода на экран сгенерированных сегментов внутри скользящей карты. Скользящая карта будет использоваться внутри нашей просмотровой функции «Правка файла фигур», которая будет показывать все геообъекты файла фигур, при этом позволяя пользователю выбирать геообъект внутри файла фигур для правки.

Приступим к реализации режима просмотра «Правка файла фигур», который мы назовем edit_shapefile(). Откройте в текстовом редакторе модуль urls.py внутри главного каталога shapeEditor и добавьте в конец списка urlpatterns следующую ниже запись:

```
url(r'^edit/(?P<shapefile_id>\d+)$',
    shapeEditor.shapefiles.views.edit_shapefile)
```

В результате любые URL-адреса, поступающие в форме /edit/N, будут перенаправляться в просмотровую функцию edit_shapefile() внутри нашего приложения shapeEditor.shapefiles.

Приступим к реализации этой функции. Откройте в текстовом редакторе модуль views.py приложения shapeEditor.shapefiles и добавьте в конец этого файла следующий ниже фрагмент:

```
def edit_shapefile(request, shapefile_id):
    try:
        shapefile = Shapefile.objects.get(id=shapefile_id)
    except Shapefile.DoesNotExist:
        return HttpResponseNotFound()

    tms_url = "http://" + request.get_host() + "/tms/"
    return render(request, "select_feature.html",
                  {'shapefile' : shapefile,
                   'tms_url' : tms_url})
```

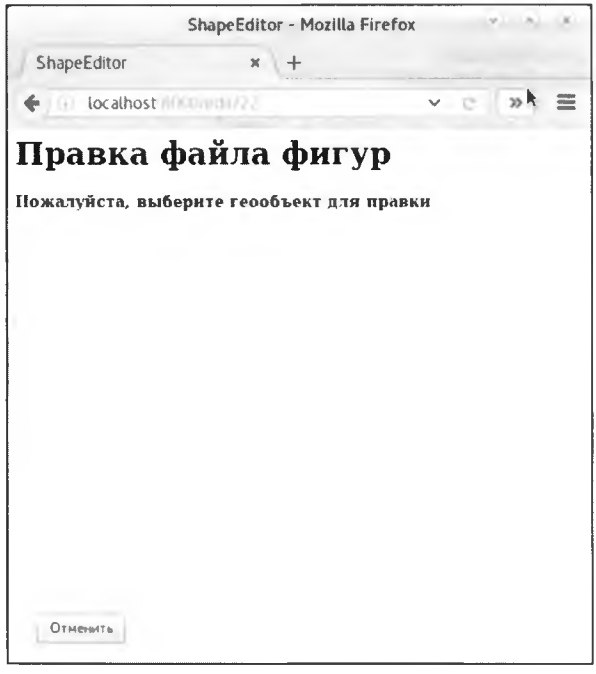
Как видите, мы получаем нужный объект Shapefile, вычисляем используемый для доступа к нашему TMS-серверу URL-адрес и передаем оба этих значения в шаблон под названием select_feature.html. Вся тяжелая работа произойдет именно в этом шаблоне.

Теперь нам нужно написать непосредственно сам шаблон. Начнем с создания нового файла под названием select_feature.html в каталоге templates приложения shapeEditor.shapefiles и введем туда следующий ниже HTML-код шаблона:

```
<html>
  <head>
    <title>ShapeEditor</title>
    <style type="text/css">
    div#map {
      width: 600px;
      height: 400px;
      border: 1px solid #ccc;
    }
    </style>
  </head>
  <body>
    <h1>Правка файла фигур</h1>
    <b>Пожалуйста, выберите геообъект для правки</b>
    <br/>
```

```
    <div id="map" class="map"></div>
    <br/>
    <div style="margin-left:20px">
      <button type="button"
              onClick='window.location="/";'>
        Отмена
      </button>
    </div>
  </body>
</html>
```

Здесь приведена всего-навсего упрощенная схема шаблона, но это дает нам хоть что-то, чтобы приступить к работе. Запустив сервер разработки Django (`python3 manage.py runserver` в окне терминала), перейдите на `http://127.0.0.1:8000` и щелкните по гиперссылке **Правка** напротив файла фигур. Вы должны увидеть упрощенный вид страницы выбора геообъекта:



Отметим, что мы создали элемент `<div>`, который будет содержать карту библиотеки OpenLayers, и используем таблицу стилей CSS, чтобы придать карте фиксированный размер и обрамление. Сама карта еще не отображена, потому что мы не написали программного кода JavaScript, необходимого для запуска библиотеки OpenLayers. Этим мы сейчас и займемся.

Добавьте в раздел `<head>` вашего шаблона следующий ниже фрагмент HTML-кода:

```
<link rel="stylesheet"
      href="http://openlayers.org/en/v3.10.1/css/ol.css"
      type="text/css">
<script src="http://openlayers.org/en/v3.10.1/build/ol.js"
        type="text/javascript">
</script>
<script src="http://code.jquery.com/jquery-2.1.4.min.js"
        type="text/javascript">
</script>
<script type="text/javascript">
  function init() {
  }
</script>
```

Кроме того, измените определение тега <body>, чтобы он выглядел нижеследующим образом:

```
<body onload="init()">
```

Отметим, что имеются три тега <script>: первый загружает библиотеку OpenLayers.js с веб-сайта http://openlayers.org, второй загружает библиотеку JQuery с http://code.jquery.com (она понадобится нам позже), и последний тег <script> будет содержать программный код создания карты на JavaScript, который мы напишем. Мы также определили функцию JavaScript под названием init(), которая будет вызываться при загрузке страницы.

Займемся реализацией функции инициализации init(). Весь приводимый ниже программный код должен быть помещен между строкой function init() { и соотвествующим символом }, который маркирует конец функции.

Мы начнем с определения переменной с именем url_template, содержащей URL-адрес, который используется для доступа к нашим сегментам:

```
var url_template = '{{ tms_url }}/1.0/{{ shapefile.id }}/{z}/{x}/
{y}.png';
```

Отметим, что мы используем специальную синтаксическую конструкцию Django {{...}}, чтобы встроить в наш шаблон URL-адрес нашего сервера сборных цифровых карт и идентификатор файла фигур. Строковые значения {z}, {x}, и {y} в конце шаблона выступают в роли местозаполнителей, которые мы заменим на масштабный уровень и координату нужного сегмента.

Далее нам нужно задать картографическую проекцию, которую библиотека OpenLayers должна использовать для этой карты:

```
var projection = ol.proj.get('EPSG:4326');
```

Напомним, что EPSG-код 4326 обозначает географические координаты широты и долготы по датуму WGS84; это именно та проекция, которая используется в наших базовых данных географической карты.

На следующем этапе мы должны определить объект **Source** библиотеки OpenLayers, который содержит источник выводимых данных географической карты. К сожалению, в библиотеке OpenLayers версии 3 отсутствует встроенный источ-

ник для серверов сборных цифровых карт (TMS), и поэтому нам придется проделать немного больше работы, чтобы получить данные географической карты с нашего сервера TMS:

```
var source = new ol.source.XYZ({
  crossOrigin: 'null',
  wrapX: false,
  projection: projection,
  tileUrlFunction: function(tileCoord) {
    var z = tileCoord[0] - 1;
    var x = tileCoord[1];
    var y = Math.pow(2, z) + tileCoord[2];
    return url_template.replace('{z}', z.toString())
                       .replace('{x}', x.toString())
                       .replace('{y}', y.toString());
  },
});
```

Как видите, мы определяем функцию `tileUrlFunction`, которая вычисляет URL-адрес заданного сегмента. Мы конвертируем масштабный уровень и прямоугольные координаты x и y, поставляемые библиотекой OpenLayers, в значения, требуемые нашим сервером TMS, и затем применяем функцию JavaScript `replace()`, чтобы заменить местозаполнители внутри нашего URL-шаблона этими расчетными значениями.

Определив источник, теперь в текущем слое можно создать объект сегмента `Tile` библиотеки OpenLayers для извлечения из этого источника сегментов сборной цифровой карты:

```
var layer = new ol.layer.Tile({source: source,
                               projection: projection,
                              });
```

Затем выполним настройку объекта `View` библиотеки OpenLayers, в котором установим для нашей карты исходный масштаб и позицию на экране:

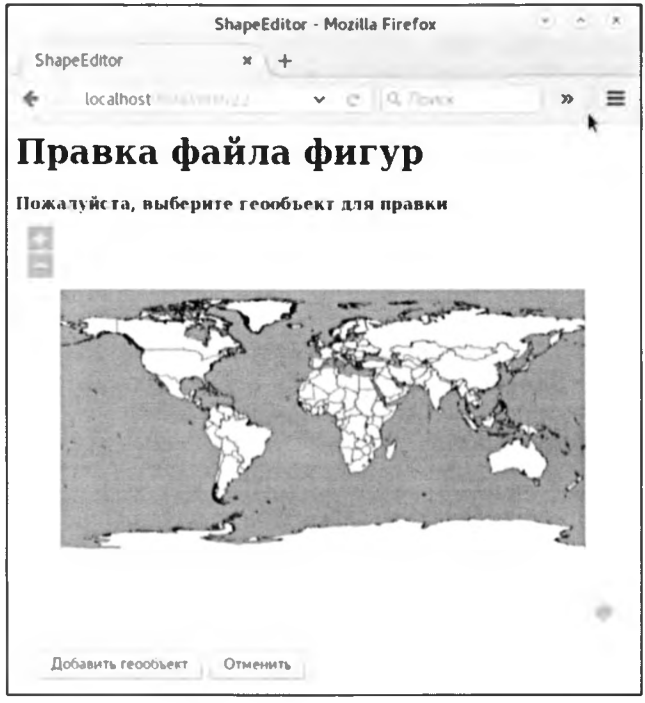
```
var view = new ol.View({center: [0, 0],
                        zoom: 1,
                        projection: projection,
                       });
```

Наконец, определим объект `Map` библиотеки OpenLayers, который выведет один-единственный слой:

```
var map = new ol.Map({target: "map",
                      layers: [layer],
                      view: view});
}
```

Отметим, что мы передаем идентификатор нашего элемента `<div>` как параметр `target`. Он поручает библиотеке OpenLayers отобразить карту внутри элемента `<div>` с этим идентификатором.

На этом реализация нашей функции `init()` завершена. Теперь протестируем то, что мы написали. Сохраните свои изменения, запустите веб-сервер Django, если он еще не работает, и направьте свой веб-браузер по адресу `http://127.0.0.1:8000`. Щелкните по гиперссылке **Правка** напротив файла фигур, который вы импортировали, и вы должны увидеть действующую скользящую карту:



Можно увеличивать и уменьшать масштаб, выполнять панорамирование в любую сторону и нажимать мышью, сколько вашей душе будет угодно. Разумеется, в действительности ничего еще не работает (кроме кнопки **Отмена**), но у нас есть скользящая карта, работающая с нашим сервером сборных цифровых карт и виджетом JavaScript библиотеки OpenLayers. Это настоящий успех!

А что, если не работает?

Если по каким-то причинам карта не отображается, то для этого может быть несколько причин. Во-первых, проверьте журнал веб-сервера Django, поскольку там протоколируются любые исключения Python. Если журнал не указывает на причину проблемы, то обратитесь к консоли сообщений об ошибках вашего веб-браузера, чтобы увидеть, нет ли каких-либо ошибок на уровне JavaScript. Поскольку мы теперь пишем программный код на JavaScript, сообщения об ошибках уже будут появляться внутри веб-браузера, а не в журнале сервера Django. В Firefox ошибки JavaScript можно просмотреть, выбрав элемент **Консоль сообщений об ошибках** из меню **Инструменты**. В других браузерах для показа ошибок JavaScript существуют аналогичные окна.

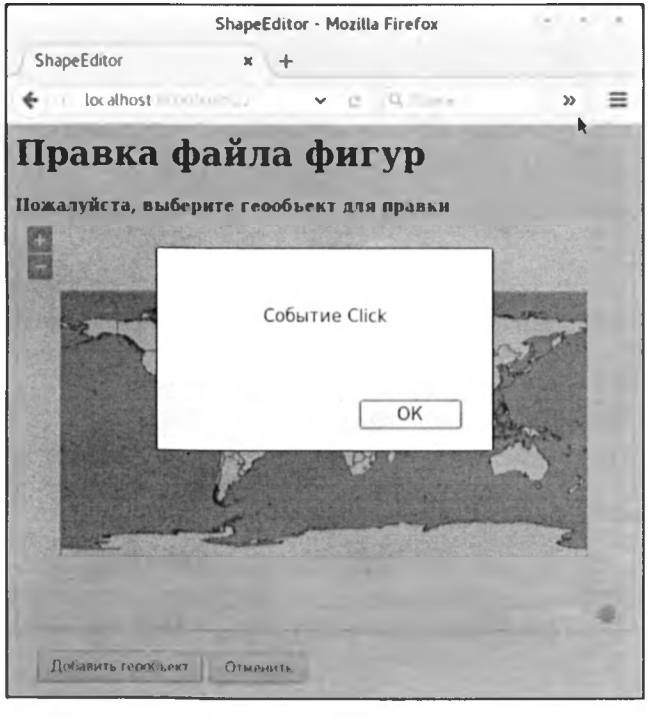
Отладка JavaScript может представлять некоторую сложность даже для людей с опытом разработки веб-приложений. Если у вас все же возникают проблемы, то указанная в ссылке статья может оказаться полезной: http://www.webmonkey.com/2010/02/javascript_debugging_for_beginners.

Перехват нажатий кнопкой мыши

Когда пользователь нажимает мышью на карте, нам нужно перехватить момент нажатия, определить прямоугольную координату, где пользователь сделал нажатие, и затем поручить серверу определить геообъект в месте нажатия (если таковой имеется). Для перехвата событий нажатия мышью мы добавим в карту функцию **прослушивания событий**. В библиотеке OpenLayers это делается легко: просто откройте в текстовом редакторе свой шаблон `select_feature.html` и добавьте в конец вашей функции `init()` следующий ниже программный код на JavaScript:

```
map.on("singleclick", function(e) {
  alert("Событие Click");
});
```

В результате всякий раз, когда пользователь нажимает на карте, будет выводиться сообщение **Событие Click**. Для проверки перезагрузите веб-страницу **Правка файла фигур** и попробуйте нажать на карту. Вы должны увидеть сообщение, как показано ниже:



Пока неплохо. Отметим, что наш обработчик нажатий перехватывает только одиночные нажатия; если на карте сделать двойное нажатие, то обработчик продолжит увеличивать масштаб.

Если ваша карта не работает, то, скорее всего, вы сделали ошибку при наборе программного кода на JavaScript. Откройте консоль JavaScript своего браузера либо окно регистрации событий и перезагрузите страницу, и в случае проблемы с вашим программным кодом на JavaScript в этом окне появится сообщение об ошибке.

Теперь займемся реализацией реального программного кода, который будет откликаться на нажатие мышью пользователя. Когда пользователь нажимает на карте, мы будем отправлять на сервер значение широты и долготы в месте нажатия, используя для этого запрос AJAX. Сервер вернет URL-адрес страницы «Правка файла фигур» для геообъекта в месте нажатия либо пустую строку, если ни на одном геообъекте нажатия не было. Если возвращается URL-адрес, то мы перенаправим веб-браузер пользователя на этот URL-адрес.

Чтобы выполнить вызов AJAX, мы воспользуемся функцией `jQuery.ajax()` библиотеки JQuery. Замените строку `alert("Событие Click");` следующим фрагментом программного кода:

```
var request = jQuery.ajax({
  url       : "/find_feature",
  data      : {shapefile_id : {{ shapefile.id }},
  latitude  : e.coordinate[1],
  longitude : e.coordinate[0]},
  success   : function(response) {
                if (response != "") {
                  window.location.href = response;
                }
              }
});
```

Как видите, функция `jQuery.ajax()` принимает URL-адрес места отправки запроса (в элементе `url`), ряд параметров для отправки на сервер (в элементе `data`) и вызов функции в случае получения ответа (в элементе `success`). Когда мы выполняем запрос, отправленные на сервер параметры будут состоять из идентификатора файла фигур и значений долготы и широты в месте нажатия на карте; когда ответ будет получен, наша функция `success` будет вызвана вместе с возвращенными сервером данными.

Библиотека JQuery фактически передает больше информации, чем просто возвращенные в функцию `success` данные, но в силу того, как работает Javascript, мы без опасений можем проигнорировать эти дополнительные параметры.

Закончив реализацию HTML-шаблона и программного кода на JavaScript, который заставляет шаблон работать, теперь напишем немного программного кода на Python, который будет фактически отыскивать геообъект в том месте, где пользователь сделал нажатие.

Реализация режима просмотра «Найти геообъект»

Теперь нам нужно написать просмотровую функцию, которая получает запрос AJAX, выясняет, на каком геообъекте было выполнено нажатие (если таковой имеется), и возвращает подходящий URL-адрес, который используется для перенаправления веб-браузера пользователя к странице «редактирования» геообъекта в месте нажатия. Чтобы это реализовать, мы воспользуемся функциями пространственных запросов модуля GeoDjango.

Начнем с того, что добавим непосредственно сам режим просмотра данных «Найти геообъект». Для этого снова откройте модуль `shapefiles.views` и добавьте следующий ниже код заготовки функции:

```
def find_feature(request):
    return HttpResponse("")
```

Возврат пустого строкового значения сообщает нашей функции обратного вызова AJAX, что ни на одном геообъекте нажатия не было. Чуть позже мы заменим заготовку на надлежащие пространственные запросы. Сейчас же нам нужно прежде всего добавить URL-шаблон, с тем чтобы входящие запросы передавались в просмотровую функцию `find_feature()`. Откройте модуль `urls.py` верхнего уровня и добавьте следующую ниже запись в список URL-шаблонов:

```
url(r'^find_feature$',
    shapeEditor.shapefiles.views.find_feature),
```

Теперь вы сможете запустить ShapeEditor, щелкнуть по гиперссылке **Правка** напротив выгруженного на сервер файла фигур, увидеть карту, показывающую различные геообъекты внутри файла фигур, и нажать где-нибудь на карте. И в ответ система, не удивляйтесь, никак не отреагирует! Это объясняется тем, что наша функция `find_feature()` возвращает пустое строковое значение, и поэтому система примет решение, что пользователь на геообъекте не нажимал, и нажатие проигнорирует.

В данном случае «отсутствие новостей» – это хорошая новость. Коль скоро не выводится никаких сообщений об ошибках ни на уровне Python, ни на уровне JavaScript, программный код AJAX работает правильно. Поэтому смело поэкспериментируйте с ним, даже при том, что ничего не происходит, просто чтобы убедиться, что ошибки в вашем коде отсутствуют. В списке получаемых сервером поступающих HTTP-запросов вы должны увидеть вызовы AJAX.

Прежде чем мы реализуем функцию `find_feature()`, сделаем шаг назад и подумаем, что означает для пользователя нажать кнопкой мыши на геометрии геообъекта. Система ShapeEditor поддерживает полный спектр возможных типов геометрий: точка Point, ломаная LineString, многоугольник Polygon, составная точка MultiPoint, составная ломаная MultiLineString, составной многоугольник MultiPolygon и коллекция геометрий GeometryCollection. Проверить, не нажал ли пользователь на геообъекте с геометрией многоугольника или с геометрией составного многоугольника, достаточно легко: мы просто проверяем, находится ли

точка нажатия внутри границ многоугольника. Однако поскольку линии и точки не имеют площади (их площадь всегда равна нулю), конкретная координата никогда не сможет быть внутри геометрии точки или ломаной линии. Можно бесконечно приближаться, но практически пользователь никогда не сможет нажать на геообъекте с геометрией точки или ломаной линии.

Рассмотрим следующий ниже пространственный запрос:

```
SELECT * FROM features WHERE ST_Contains(feature.geometry, clickPt)
```

Он не будет работать, потому что место нажатия не может быть внутри геометрии точки или ломаной. Вместо этого мы должны допустить ситуацию, когда пользователь нажимает не внутри геообъекта, а близко к нему. Для этого мы рассчитаем радиус поиска в единицах измерения карты и затем задействуем функцию пространственного запроса DWithin, чтобы найти все геообъекты в пределах заданного радиуса поиска вокруг точки нажатия.

К сожалению, здесь есть одна проблема. В *главе 8 «Работа с пространственными данными»* мы использовали пространственный запрос ST_DWithin, чтобы вычислить все геообъекты в пределах заданного количества метров от заданной точки. Мы могли это выполнить, потому что геообъекты, с которыми мы работали, хранились в столбце PostGIS **Geography**. Однако в данном случае наши пространственные данные хранятся в столбце **Geometry**, а не Geography. Это означает, что радиус поиска, который мы передаем в функцию пространственного запроса DWithin веб-платформы Django, должен исчисляться в градусах широты и долготы, а не в метрах.

Увы, но в приложении ShapeEditor мы не сможем использовать столбцы geography из-за ограничений, накладываемых на то, как они работают. Вместо этого нам придется вычислять радиус поиска как число градусов.

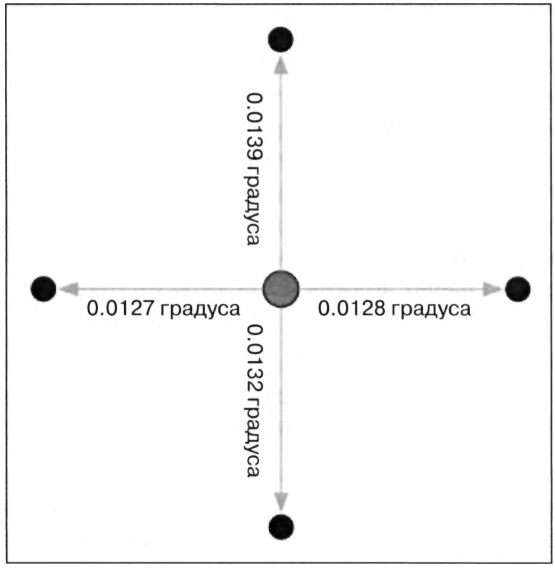
Учитывая, что пользователь может нажать на карте поверхности Земли в любом месте, нам придется рассчитывать радиус поиска на основе места нажатия. Это объясняется тем, что связь между прямоугольными координатами и фактическими расстояниями на поверхности Земли может значительно отличаться в зависимости от места, где пользователь нажал мышью: 1° на экваторе равен расстоянию 111 км, в то время как 1° в Швеции – всего половине этого расстояния.

С целью обеспечения надежного радиуса поиска по всей поверхности планеты мы будем пользоваться библиотекой PROJ.4, которая позволит рассчитать радиус поиска в единицах измерения карты при наличии места нажатия и нужного линейного расстояния. Добавим эту функцию в наш модуль shared.utils:

```
def calc_search_radius(latitude, longitude, distance):
    geod = pyproj.Geod(ellps="WGS84")

    x,y,angle = geod.fwd(longitude, latitude, 0, distance)
    radius = y-latitude

    x,y,angle = geod.fwd(longitude, latitude, 90, distance)
    radius = max(radius, x-longitude)
```

```
    x,y,angle = geod.fwd(longitude, latitude, 180, distance)
    radius = max(radius, latitude-y)

    x,y,angle = geod.fwd(longitude, latitude, 270, distance)
    radius = max(radius, longitude-x)

    return radius
```

Эта функция вычисляет расстояние в единицах измерения карты для заданного линейного расстояния, измеряемого в метрах. Она вычисляет географические координаты широты и долготы для четырех точек непосредственно к северу, югу, востоку и западу от исходного местоположения и на расстоянии заданного количества метров от этой точки. Затем она вычисляет разность в широте или долготе между исходным местом и конечной точкой:



Наконец, она выбирает максимальную из этих разностей и возвращает ее как радиус поиска, который измеряется в градусах широты или долготы.

Поскольку наш модуль `utils.py` теперь использует библиотеку Python `pyproj`, добавьте в начало этого модуля следующий ниже оператор импорта:

```
import pyproj
```

Написав функцию `calc_search_radius()`, мы теперь можем воспользоваться пространственным запросом `DWithin` веб-платформы Django, чтобы идентифицировать все геообъекты рядом с местом нажатия. В GeoDjango весь процесс сводится к использованию функции `filter()`, которая создает пространственный запрос, как показано в следующем ниже примере:

```
query = Feature.objects.filter(geometry__dwithin=(pt, radius))
```

В результате будет получен набор объектов Feature, которые соответствуют заданным критериям. GeoDjango умело добавляет поддержку пространственных запросов во встроенный в веб-платформу Django функционал фильтрации; в этом случае параметр geometry__dwithin=(pt, radius) поручает модулю GeoDjango выполнить пространственный запрос DWithin по полю geometry внутри объекта Feature, используя два предоставленных параметра.

Таким образом, модуль GeoDjango выполнит трансляцию этого оператора в пространственный запрос базы данных, который выглядит примерно так:

```
SELECT * from feature WHERE ST_DWithin(geometry, pt, radius)
```

Отметим, что именованный параметр geometry__dwithin содержит два символа подчеркивания; в Django двойное подчеркивание используется для различения имени поля от имени функции фильтрации.

Зная об этом и имея реализованную функцию utils.calc_search_radius(), мы можем, наконец, реализовать нашу просмотровую функцию find_feature(). Откройте в текстовом редакторе модуль shapefile.views и замените тело функции find_feature() следующим ниже программным кодом:

```
def find_feature(request):
    try:
        shapefile_id = int(request.GET['shapefile_id'])
        latitude = float(request.GET['latitude'])
        longitude = float(request.GET['longitude'])

        shapefile = Shapefile.objects.get(id=shapefile_id)
        pt = Point(longitude, latitude)
        radius = utils.calc_search_radius(latitude, longitude, 100)

        if shapefile.geom_type == "Point":
            query = Feature.objects.filter(
                geom_point__dwithin=(pt, radius))
        elif shapefile.geom_type in ["LineString", "MultiLineString"]:
            query = Feature.objects.filter(
                geom_multilinestring__dwithin=(pt, radius))
        elif shapefile.geom_type in ["Polygon", "MultiPolygon"]:
            query = Feature.objects.filter(
                geom_multipolygon__dwithin=(pt, radius))
        elif shapefile.geom_type == "MultiPoint":
            query = Feature.objects.filter(
                geom_multipoint__dwithin=(pt, radius))
        elif shapefile.geom_type == "GeometryCollection":
            query = feature.objects.filter(
                geom_geometrycollection__dwithin=(pt, radius))
        else:
            print("Неподдерживаемая геометрия: " + shapefile.geom_type)
            return HttpResponse("")

        if query.count() != 1:
```

```
            return HttpResponse("")

        feature = query[0]
        return HttpResponse("/edit_feature/" +
                            str(shapefile_id)+"/"+str(feature.id))
    except:
        traceback.print_exc()
        return HttpResponse("")
```

Здесь полно всего, поэтому рассмотрим все по порядку. Прежде всего мы завернули весь наш программный код в оператор `try...except`:

```
def find_feature(request):
    try:
        ...
    except:
        traceback.print_exc()
        return HttpResponse("")
```

Это тот же самый прием, который мы использовали при реализации сервера сборных цифровых карт; он подразумевает, что любые ошибки Python в вашем коде будут отображены в журнале веб-сервера, а функция AJAX изящно вернет результат вместо аварийного отказа.

Затем мы извлекаем параметры переданного запроса, конвертируя их из строковых значений в числа, загружаем нужный объект `Shapefile`, создаем из координат в месте нажатия объект `Point` модуля GeoDjango и вычисляем радиус поиска в градусах:

```
    shapefile_id = int(request.GET['shapefile_id'])
    latitude = float(request.GET['latitude'])
    longitude = float(request.GET['longitude'])

    shapefile = Shapefile.objects.get(id=shapefile_id)
    pt = Point(longitude, latitude)
    radius = utils.calc_search_radius(latitude, longitude, 100)
```

Отметим, что мы используем фиксированный («зашитый») радиус поиска в 100 метров: этого достаточно, чтобы, нажимая рядом с геообъектом, имеющим геометрию точки или линии, который не настолько большой, что пользователь может случайно нажать на неправильном геообъекте, позволить пользователю его выбрать.

Закончив с этим, мы теперь готовы выполнить пространственный запрос. Поскольку наш объект `Feature` имеет отдельные поля для каждого типа геометрии, мы должны сконструировать запрос на основе типа геометрии:

```
    if shapefile.geom_type == "Point":
        query = Feature.objects.filter(
            geom_point__dwithin=(pt, radius))
    elif shapefile.geom_type in ["LineString", "MultiLineString"]:
        query = Feature.objects.filter(
```

```
            geom_multilinestring__dwithin=(pt, radius))
    elif shapefile.geom_type in ["Polygon", "MultiPolygon"]:
        query = Feature.objects.filter(
            geom_multipolygon__dwithin=(pt, radius))
    elif shapefile.geom_type == "MultiPoint":
        query = Feature.objects.filter(
            geom_multipoint__dwithin=(pt, radius))
    elif shapefile.geom_type == "GeometryCollection":
        query = feature.objects.filter(
            geom_geometrycollection__dwithin=(pt, radius))
    else:
        print("Неподдерживаемая геометрия: " + shapefile.geom_type)
    return HttpResponse("")
```

В каждом случае мы выбираем соответствующее поле геометрии и используем `__dwithin`, чтобы выполнить пространственный запрос на подходящем поле в объекте `Feature`.

После того как соответствующий пространственный запрос создан, мы просто проверяем, что запрос возвратил ровно один объект `Feature`. Если нет, то мы возвращаем пустое строковое значение в функцию обратного вызова обработчика AJAX, сообщая ей, что пользователь на геообъекте не нажимал:

```
if query.count() != 1:
    return HttpResponse("")
```

Если имеется строго один совпавший геообъект, то в нашем распоряжении геообъект в месте нажатия кнопкой мыши, и мы воспользуемся им, чтобы создать URL-адрес, который перенаправит веб-браузер пользователя по URL-адресу страницы «Правка геообъекта» с идентификатором геообъекта в месте нажатия:

```
feature = query[0]
return HttpResponse("/edit_feature/" +
                    str(shapefile_id)+"/"+str(feature.id))
```

Набрав весь этот программный код, добавьте в начало модуля `views.py` следующие ниже операторы импорта:

```
import traceback
from django.contrib.gis.geos import Point
from shapeEditor.shared.models import Feature
from shapeEditor.shared import utils
```

На этом наша просмотровая функция `find_feature()` завершена. Сохраните свои изменения, запустите веб-сервер Django, в случае если он еще не работает, и попробуйте нажать на геообъекте файла фигур. Если вы нажмете на океане, то ничего не произойдет – но если вы нажмете на геообъекте, то вы должны увидеть, что веб-браузер будет перенаправлен по URL-адресу следующего ниже вида:

```
http://127.0.0.1:8000/edit_feature/X/Y
```

Здесь X – это идентификатор файла фигур и Y – идентификатор геообъекта в месте нажатия. Конечно, на данном этапе вы получите ошибку **«Страница не найдена»**, потому что эту страницу вы еще не написали. Но, по крайней мере, можно нажать на геообъекте, чтобы его выбрать, что свидетельствует о преодолении важного этапа разработки веб-приложения ShapeEditor. Поздравляем!

Правка геообъектов

Зная, какой геообъект пользователь хочет отредактировать, теперь наша следующая задача состоит в том, чтобы реализовать непосредственно саму страницу «Правка геообъекта». Для этого нам придется создать пользовательскую форму с единственным полем ввода под названием `geometry`, в котором используется подходящий виджет редактирования карты для правки геометрии геообъекта.

Процесс создания этой формы немного запутан из-за того, что нам приходится на лету создавать новый подкласс `django.contrib.gis.forms.Form`, чтобы учесть различные типы геометрий, которые могут встретиться при редактировании. Поместим решение этой сложной задачи в новую функцию внутри модуля `shared.utils`, которую мы назовем `get_map_form()`.

Откройте модуль `utils.py` в текстовом редакторе и наберите следующий ниже фрагмент кода:

```
def get_map_form(shapefile):
    geometry_field = calc_geometry_field(shapefile.geom_type)

    if geometry_field == "geom_multipoint":
        field = forms.MultiPointField
    elif geometry_field == "geom_multilinestring":
        field = forms.MultiLineStringField
    elif geometry_field == "geom_multipolygon":
        field = forms.MultiPolygonField
    elif geometry_field == "geom_geometrycollection":
        field = forms.GeometryCollectionField
    else:
        raise RuntimeError("Неподдерживаемое поле: " + geometry_field)

    widget = forms.OpenLayersWidget()

    class MapForm(forms.Form):
        geometry = field(widget=widget)

    return MapForm
```

Вам также придется добавить в начало файла следующий ниже оператор импорта:

```
from django.contrib.gis import forms
```

Функция `get_map_form()` выбирает тип поля для редактирования на основе атрибута `geom_type` в файле фигур. Например, если файл фигур содержит геометрии

точки или составной точки, то для редактирования данных файла фигур мы выбираем класс MultiPointField. После того как тип поля выбран, мы динамически создаем новый подкласс django.contrib.gis.forms.Form, который для редактирования данных геометрии использует экземпляр этой формы вместе с виджетом OpenLayersWidget.

Отметим, что функция get_map_form() возвращает не экземпляр класса MapForm, а сам *класс*; мы воспользуемся классом MapForm для создания соответствующих экземпляров, поскольку они нам понадобятся в дальнейшем.

Отложив эту функцию в сторону, теперь мы можем реализовать остальную часть режима просмотра «Правка геообъекта». Начнем с настройки URL-адреса режима просмотра данных: откройте модуль urls.py и добавьте в список URL-шаблонов нижеследующую запись:

```
url(r'^edit_feature/(?P<shapefile_id>\d+)/(?P<feature_id>\d+)$',
    shapeEditor.shapefiles.views.edit_feature),
```

Теперь мы готовы реализовать непосредственно саму просмотровую функцию. Откройте модуль shapefiles.views в текстовом редакторе и начните определять функцию edit_feature():

```
def edit_feature(request, shapefile_id, feature_id):
    try:
        shapefile = Shapefile.objects.get(id=shapefile_id)
    except ShapeFile.DoesNotExist:
        return HttpResponseNotFound()

    try:
        feature = Feature.objects.get(id=feature_id)
    except Feature.DoesNotExist:
        return HttpResponseNotFound()
```

Пока все довольно прямолинейно: мы загружаем объект Shapefile для текущего файла фигур и объект Feature для редактируемого геообъекта. Далее нам нужно загрузить в память список атрибутов этого геообъекта, чтобы показать их пользователю:

```
  attributes = []
  for attr_value in feature.attributevalue_set.all():
      attributes.append([attr_value.attribute.name,
                         attr_value.value])
  attributes.sort()
```

И с этого места все становится интереснее. Нам нужно создать объект Form веб-платформы Django (фактически экземпляр класса MapForm, динамически создаваемого написанной нами ранее функцией get_map_form()) и использовать этот экземпляр формы, чтобы показать геообъект, который будет отредактирован. Когда форма будет отправлена, мы извлечем обновленную геометрию и снова сохраним ее в объекте Feature, после чего перенаправим пользователя назад на страницу «Правка файла фигур» для выбора другого геообъекта.

Как мы уже убедились, когда создавали форму «Импорт файла фигур», базовая идиома веб-платформы Django в отношении обработки форм состоит в следующем:

```
if request.method == "GET":
    form = MyForm()
    return render(request, "template.html",
                  {'form' : form})
elif request.method == "POST":
    form = MyForm(request.POST)
    if form.is_valid():
       # Извлечь и сохранить содержимое формы тут...
       return HttpResponseRedirect("/somewhere/else")
    return render(request, "template.html",
                  {'form' : form})
```

При первом показе формы свойство `request.method` будет иметь значение `GET`. В этом случае мы создаем новый объект формы и выводим форму как составную часть HTML-шаблона. Когда пользователь отправит форму на сервер, свойство `request.method` будет иметь значение `POST`. В этом случае создается новый объект формы, который увязан с предоставленными параметрами `POST`. Затем содержимое формы проверяется, и, если оно допустимо, оно сохраняется, и пользователь перенаправляется на некую другую страницу. Если форма не допустима, то она будет выведена повторно вместе с подходящим сообщением об ошибке.

Посмотрим, как эта идиома используется режимом просмотра данных «Правка геообъекта». Добавьте в конец своей новой просмотровой функции следующий ниже фрагмент:

```
geometry_field = \
    utils.calc_geometry_field(shapefile.geom_type)
form_class = utils.get_map_form(shapefile)

if request.method == "GET":
    wkt = getattr(feature, geometry_field)
    form = form_class({'geometry' : wkt})

    return render(request, "edit_feature.html",
                  {'shapefile' : shapefile,
                   'form' : form,
                   'attributes' : attributes})
elif request.method == "POST":
    form = form_class(request.POST)
    try:
        if form.is_valid():
```

```
            wkt = form.cleaned_data['geometry']
            setattr(feature, geometry_field, wkt)
            feature.save()
            return HttpResponseRedirect("/edit/" +
                                        shapefile_id)
    except ValueError:
        pass

    return render(request, "edit_feature.html",
                  {'shapefile' : shapefile,
                   'form' : form,
                   'attributes' : attributes})
```

Как видите, мы вызываем функцию `utils.get_map_form()`, чтобы создать новый подкласс `django.forms.Form`, который будет использован при редактировании геометрии геообъекта. Мы также вызываем функцию `utils.calc_geometry_field()`, чтобы извлечь поле в объекте `Feature` для правки.

Остальная часть этой функции в значительной степени подчинена идиоме Django обработки форм. Единственно стоит отметить то, что мы получаем и устанавливаем значение поля геометрии (применяя, соответственно, функции `getattr()` и `setattr()`), используя формат WKT. Модуль GeoDjango рассматривает поля геометрии так, как если бы они были символьными полями, в которых геометрия содержится в формате WKT. Затем программный код на JavaScript модуля GeoDjango берет эти данные WKT (которые хранятся в скрытом поле `geometry` формы) и передает их библиотеке OpenLayers для вывода в виде векторной геометрии. Библиотека OpenLayers предоставляет пользователю возможность эту векторную геометрию отредактировать, после чего обновленная геометрия снова сохраняется в формате WKT в скрытом поле `geometry`. Затем мы извлекаем текст WKT обновленной геометрии и снова сохраняем его в объекте `Feature`.

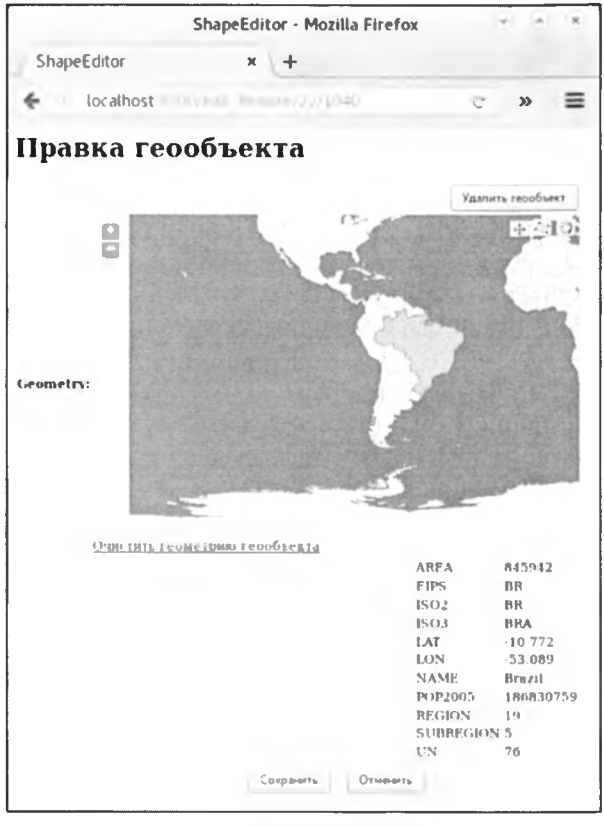
Это все, что необходимо знать о просмотровой функции `edit_feature()`. Теперь создадим шаблон, используемый этим режимом просмотра данных. Внутри каталога `templates` приложения `shapefiles` создайте новый файл под названием `edit_feature.html` и введите туда следующий ниже текст:

```
<html>
  <head>
    <title>ShapeEditor</title>
    {{ form.media }}
  </head>
  <body>
    <h1>Правка геообъекта</h1>
    <form method="POST" action="">
```

```
      <table>
        {{ form.as_table }}
        <tr>
          <td></td>
          <td align="right">
            <table>
{% for attr in attributes %}
              <tr>
                <td>{{ attr.0 }}</td>
                <td>{{ attr.1 }}</td>
              </tr>
{% endfor %}
            </table>
          </td>
        </tr>
        <tr>
          <td></td>
          <td align="center">
            <input type="submit" value="Сохранить"/>
             
            <button type="button" onClick='window.location="/edit/{{shapefile.id }}";'>
              Отменить
            </button>
          </td>
        </tr>
      </table>
    </form>
  </body>
</html>
```

В этом шаблоне для отображения формы используется HTML-таблица, и для визуализации формы в виде строк HTML-таблицы вызывается функция шаблона `form.as_table`. Затем в дополнительных строках внутри этой таблицы мы показываем список атрибутов геообъекта и в заключение в основание формы выносим кнопки **Сохранить** и **Отменить**.

Закончив писать этот программный код, мы в конечном итоге получили возможность редактировать геообъекты внутри веб-приложения ShapeEditor:



Внутри этого редактора можно использовать ряд встроенных в GeoDjango функций редактирования геометрий:

- можно, нажав на инструменте **Правка геометрии** (), выбрать геообъект для правки;
- можно, нажав на инструменте **Добавление геометрии** (), начать отрисовку новой геометрии;
- после выбора геометрии можно нажать на темной окружности и ее перетащить, чтобы переместить конечные точки отрезка;
- после выбора геометрии можно нажать на светлой окружности и разделить существующий отрезок на два, создав новую точку, которую затем можно перетащить;
- если удерживать мышь на темной окружности, то для удаления этой точки можно нажать клавишу **Удалить** (или клавишу D). Отметим, что это будет работать, только если геометрия имеет более трех точек;
- можно нажать на гиперссылке **Delete all Features** (Удалить все геообъекты), чтобы удалить геометрии текущих геообъектов. Чуть позже мы рассмотрим эту гиперссылку более подробно.

Закончив редактировать геообъект, можно нажать кнопку **Сохранить**, чтобы сохранить отредактированные геообъекты, или кнопку **Отмена**, чтобы отклонить все изменения.

Несмотря на то что все это хорошо работает, все же есть один несколько неудобный момент: для удаления из карты геометрии GeoDjango позволяет пользователю применять гиперссылку **Очистить геометрию геообъекта**. Учитывая, что мы сейчас внутри файла фигур редактируем всего один геообъект, по непонятной причине эта гиперссылка носит довольно запутывающее название: на самом деле если перейти по гиперссылке, то будет удален не сам геообъект, а его *геометрии*. Следует поменять текст этой гиперссылки на что-то более вразумительное.

В скачанном вами экземпляре веб-платформы Django перейдите в каталог `contrib/gis/templates/gis`. В этом каталоге имеется файл под названием `openlayers.html`. Сделайте копию этого файла и переместите его в каталог `templates` своего приложения `shapefiles`, при этом переименуйте его в `openlayers-custom.html`.

Откройте свою копию этого файла, и ближе к концу файла вы найдете текст `Delete all Features`. Поменяйте его на `Очистить геометрию геообъекта` и сохраните ваши изменения.

Пока все идет неплохо. Теперь мы должны указать виджету редактирования GeoDjango, чтобы он использовал нашу собственную версию файла `openlayers.html`. Для этого откройте в текстовом редакторе свой модуль `utils.py` и отыщите ваше определение функции `get_map_form()`. Добавьте следующий ниже код сразу после строки с `widget = forms.OpenLayersWidget()`:

```
widget.template_name = "openlayers-custom.html"
```

Если затем попробовать отредактировать геообъект, то вы увидите, что используется ваша адаптированная версия файла `openlayers.html`:

Как видите, можно вносить различные изменения во внешний вид и функциональность виджета редактирования геометрий, причем это делается путем замены шаблона, передачи параметров в виджет при его инициализации и создания своего собственного подкласса класса `BaseGeometryWidget`. Если вы хотите взглянуть на то, что еще можно сделать, обратитесь к модулям в каталоге `django.contrib.gis`.

Добавление геообъектов

Далее мы приступим к реализации функционала добавления нового геообъекта. Для этого мы поместим кнопку **Добавить геообъект** в режим просмотра данных «Правка файла фигур». В результате нажатия на этой кнопке ваш браузер обратится к URL-адресу «Правка геообъекта», но без идентификатора геообъекта. За-

тем мы изменим режим просмотра данных «Правка геообъекта» так, чтобы при отсутствующем идентификаторе геообъекта создавался новый объект Feature.

Откройте модуль views.py приложения shapefiles, отыщите функцию edit_shapefile() и добавьте к этой функции следующие ниже выделенные строки:

```
def edit_shapefile(request, shapefile_id):
    try:
        shapefile = Shapefile.objects.get(id=shapefile_id)
    except Shapefile.DoesNotExist:
        raise Http404

    tms_url = "http://"+request.get_host()+"/tms/"
    find_feature_url = "http://" + request.get_host() \
                     + "/editor/find_feature"
    add_feature_url = "http://" + request.get_host() \
                    + "/edit_feature/" + str(shapefile_id)

    return render(request, "select_feature.html",
                  {'shapefile' : shapefile,
                   'find_feature_url' : find_feature_url,
                   'add_feature_url' : add_feature_url,
                   'tms_url' : tms_url})
```

Затем откройте в текстовом редакторе шаблон select_feature.html и добавьте в тело шаблона следующие ниже выделенные строки:

```
<body onload="init()">
  <h1>Правка файла фигур</h1>
  <b>Пожалуйста, выберите геообъекты для правки</b>
  <br/>
  <div id="map" class="map"></div>
  <br/>
  <div style="margin-left:20px">
    <button type="button"
            onClick='window.location="{{ add_feature_url }}";'>
      Добавить геообъект
    </button>
   <button type="button"
           onClick='window.location="/";'>
      Отменить
    </button>
  </div>
</body>
```

В результате на странице «Выбор геообъекта» будет размещена кнопка **Добавить геообъект**. Нажатие на этой кнопке вызовет URL-адрес http://127.0.0.1:8000/edit_feature/N (где N – идентификатор текущего файла фигур). Как видите, вместо идентификатора редактируемого геообъекта этот URL-адрес предоставляет идентификатор нужного файла фигур. Это объясняется тем, что мы не редактируем существующий геообъект, а добавляем новый.

Далее нам нужно добавить URL-шаблон, который поддерживает этот URL-адрес. Откройте модуль `urls.py` и добавьте к списку URL-шаблонов следующую ниже запись:

```
url(r'^edit_feature/(?P<shapefile_id>\d+)$',
    shapeEditor.shapefiles.views.edit_feature),
```

Затем вернитесь в `views.py` и измените определение функции `edit_feature()`, чтобы оно выглядело следующим образом:

```
def edit_feature(request, shapefile_id, feature_id=None):
```

Отметим, что параметр `feature_id` теперь необязательный. Далее отыщите следующий блок кода:

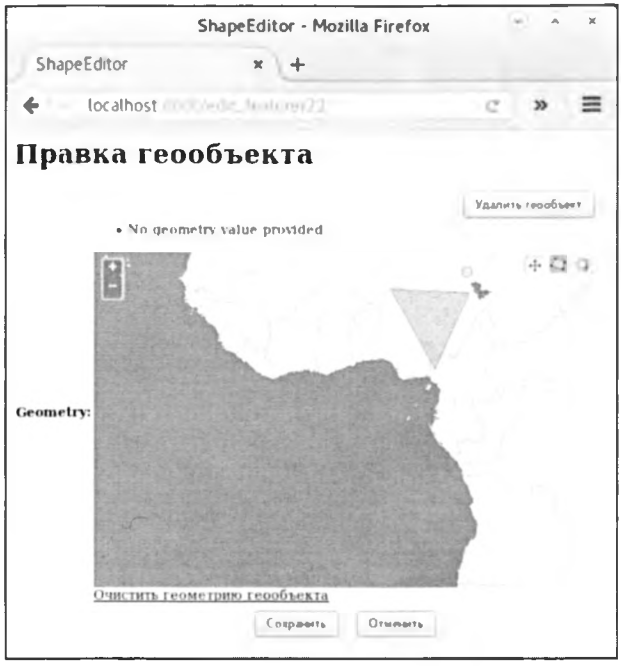
```
try:
    feature = Feature.objects.get(id=feature_id)
except Feature.DoesNotExist:
    return HttpResponseNotFound()
```

Необходимо поменять этот блок на следующий ниже:

```
if feature_id == None:
    feature = Feature(shapefile=shapefile)
else:
    try:
        feature = Feature.objects.get(id=feature_id)
    except Feature.DoesNotExist:
        return HttpResponseNotFound()
```

В результате, в случае если значение `feature_id` не определено, будет создан новый объект `Feature`, но если идентификатор геообъекта будет недопустимым, то по-прежнему результата не будет.

Имея в распоряжении все эти изменения, вы сможете добавлять в файл фигур новый геообъект. Пойдем дальше и протестируем это: запустите веб-сервер Django, в случае если он еще не работает, и нажмите на гиперссылке **Правка** напротив импортированного файла форм. Затем нажмите на гиперссылке **Добавить новый геообъект** и попробуйте создать новый геообъект. Новый геообъект должен появиться в режиме просмотра «Выбор геообъекта»:



Удаление геообъектов

Далее нам нужно разрешить пользователю удалять существующий геообъект. Для этого мы добавим кнопку **Удалить геообъект** в режим просмотра данных «Правка гообъекта». В результате нажатия на этой кнопке пользователь будет перенаправлен в режим просмотра данных «Удаление геообъекта» для этого геообъекта.

Откройте в текстовом редакторе шаблон `edit_feature.html` и добавьте в раздел `<form>` шаблона следующие ниже выделенные строки:

```
<form method="POST" action="">
  <table>
    <tr>
      <td></td>
      <td align="right">
        <input type="submit" name="delete"
               value="Удалить геообъект"/>
      </td>
    </tr>
    {{ form.as_table }}
    ...
```

Отметим, что для этой кнопки мы использовали элемент <input type="submit">. В результате форма будет отправлена вместе с дополнительным параметром POST под названием delete. Теперь вернитесь в модуль shapefiles.views и добавьте в начало функции edit_feature() следующий ниже фрагмент:

```
if request.method == "POST" and "delete" in request.POST:
    return HttpResponseRedirect("/delete_feature/" +
                                shapefile_id+"/"+feature_id)
```

Далее нам нужно реализовать режим просмотра данных «Удаление геообъекта». Откройте модуль urls.py верхнего уровня и добавьте в список URL-шаблонов нижеследующее:

```
url(r'^delete_feature/(?P<shapefile_id>\d+)/(?P<feature_id>\d+)$',
    shapeEditor.shapefiles.views.delete_feature),
```

Далее в каталоге shapeEditor/shapefiles/templates создайте новый файл с именем delete_feature.html и введите туда следующий ниже текст:

```
<html>
  <head>
    <title>ShapeEditor</title>
  </head>
  <body>
    <h1>Удалить геообъект</h1>
    <form method="POST">
      Вы уверены, что хотите удалить этот геообъект?
      <p/>
      <button type="submit" name="confirm"
              value="1">Удалить</button>
       
      <button type="submit" name="confirm"
              value="0">Отменить</button>
    </form>
  </body>
</html>
```

Эта простая HTML-форма служит для подтверждения удаления. После отправки формы, в случае если пользователь желает удалить геообъект, параметр POST с именем confirm получит значение 1. Теперь реализуем режим просмотра данных, который использует этот шаблон. Откройте модуль shapefiles.views и добавьте следующую ниже новую просмотровую функцию:

```
def delete_feature(request, shapefile_id, feature_id):
    try:
        feature = Feature.objects.get(id=feature_id)
    except Feature.DoesNotExist:
        return HttpResponseNotFound()

    if request.method == "POST":
```

```
        if request.POST['confirm'] == "1":
            feature.delete()
        return HttpResponseRedirect("/edit/" +
                                    shapefile_id)
    return render(request, "delete_feature.html")
```

Как видите, удаление геообъектов выполняется довольно прямолинейно.

Удаление файлов фигур

Осталось реализовать заключительный элемент функциональности – режим просмотра данных «Удалить файл фигур». Он позволит пользователю удалять весь выгруженный на сервер файл фигур. В своей основе этот процесс аналогичен процессу удаления геообъектов: на главной странице у нас уже есть гиперссылка **Удалить**, и поэтому нам всего лишь осталось реализовать лежащий в основе режим просмотра данных.

Перейдите в модуль `urls.py` верхнего уровня и добавьте в список URL-шаблонов следующую ниже запись:

```
url(r'^delete/(?P<shapefile_id>\d+)$',
    shapeEditor.shapefiles.views.delete_shapefile),
```

Затем откройте в текстовом редакторе модуль `shapefiles.views` и добавьте следующую ниже новую просмотровую функцию:

```
def delete_shapefile(request, shapefile_id):
    try:
        shapefile = Shapefile.objects.get(id=shapefile_id)
    except Shapefile.DoesNotExist:
        return HttpResponseNotFound()

    if request.method == "GET":
        return render(request, "delete_shapefile.html",
                      {'shapefile' : shapefile})
    elif request.method == "POST":
        if request.POST['confirm'] == "1":
            shapefile.delete()
        return HttpResponseRedirect("/")
```

Отметим, что мы передаем в шаблон объект `Shapefile`. Это объясняется тем, что нам нужно отобразить на странице подтверждения некоторую информацию о файле фигур.

Напомним, что метод `shapefile.delete()` не просто удаляет непосредственно сам объект `Shapefile`; он также удаляет все объекты, связанные с объектом `Shapefile` полями `ForeignKey`. Это означает, что один вызов `shapefile.delete()` также удалит все связанные с этим объектом `Shapefile` объекты `Attribute`, `Feature` и `AttributeValue`.

Наконец, создайте новый шаблон под названием `delete_shapefile.html` и наберите туда следующий ниже текст:

```
<html>
  <head>
    <title>ShapeEditor</title>
  </head>
  <body>
    <h1>Удалить файл фигур</h1>
    <form method="POST">
      Вы уверены, что желаете удалить
      "{{ shapefile.filename }}" файл фигур?
      <p/>
      <button type="submit" name="confirm"
              value="1">Удалить</button>
       
      <button type="submit" name="confirm"
              value="0">Отменить</button>
    </form>
  </body>
</html>
```

Теперь вы можете нажать на гиперссылке **Удалить** и удалить файл фигур. Можно смело это проверить, так как файл фигур, если нужно, можно всегда импортировать повторно.

Использование системы ShapeEditor

Примите наши поздравления! Вы только что закончили реализацию последнего компонента системы ShapeEditor, и теперь у вас есть полнофункциональное действующее геопространственное приложение, созданное при помощи географического модуля GeoDjango. При помощи ShapeEditor можно импортировать файлы фигур, просматривать их геообъекты и атрибуты, вносить изменения в геометрии геообъектов, добавлять и удалять геообъекты и затем снова экспортировать файлы фигур.

Это, конечно же, полезное приложение. Даже если у вас отсутствует полноценная геоинформационная система (ГИС), теперь у вас есть возможность вносить быстрые и легкие изменения в содержание файла фигур, используя для этого систему ShapeEditor. И разумеется, ShapeEditor – это замечательная отправная точка для разработки своих собственных геопространственных приложений.

Дальнейшие усовершенствования и улучшения

Как и с любым новым приложением, существует целый ряд приемов, позволяющих улучшить работу системы ShapeEditor. Вот несколько примеров:

- добавление функционала, связанного с регистрацией пользователя и входом в систему, с тем чтобы каждый пользователь имел свой собственный

частный набор файлов фигур, а не как сейчас, когда каждый пользователь видит один и тот же список всех выгруженных на сервер файлов фигур;
- добавление функционала, связанного с редактированием значений атрибутов геообъекта;
- использование таблиц стилей CSS и, может быть, библиотеки пользовательского интерфейса, такой как Bootstrap, с целью улучшения внешнего вида веб-страниц системы;
- использование базовой карты с более высоким разрешением. Очевидным кандидатом для этого могла бы стать база данных береговых линий GSHHS с высоким разрешением;
- добавление кэша сегментов для нашего сервера TMS;
- использование JavaScript для добавления всплывающего сообщения **Пожалйста, подождите**, которое появляется во время импортирования или экспортирования файла фигур;
- улучшение возможности многократного использования базы исходного кода системы ShapeEditor. Мы сосредоточили все наше внимание на изучении методов использования модуля GeoDjango для создания действующей системы, но при подходящей переработке проекта программный код можно сделать намного более универсальным, с тем чтобы его можно было использовать и в других приложениях.

Смело вносите эти улучшения; вы узнаете намного больше о географическом модуле GeoDjango и о разработке геоприложений в целом. И несомненно, во время работы с системой ShapeEditor у вас, скорее всего, появится свой собственный список того, что требует улучшения.

Заключение

В этой главе мы закончили реализацию высокотехнологичного геопространственного веб-приложения на основе модуля GeoDjango, расширения PostGIS, библиотек Mapnik, OGR и pyproj. Это приложение полезно само по себе и вместе с тем служит трамплином для разработки своих собственных геопространственных веб-приложений.

Мы узнали, что можем легко создать свой собственный сервер сборных цифровых карт, используя для этого библиотеку Mapnik и расширение GeoDjango. Мы научились использовать библиотеку OpenLayers на наших собственных вебстраницах и увидели, каким образом эта библиотека может быть задействована в работе с нашим сервером сборных цифровых карт. Мы научились при помощи библиотеки OpenLayers перехватывать нажатия кнопки мыши и использовать функциональность AJAX библиотеки JQuery с целью отправки запросов на сервер для обработки.

Далее мы узнали, как использовать библиотеку PROJ.4 для расчета радиуса поиска, измеряемого в градусах широты и долготы, и как использовать имеющиеся в GeoDjango функции запроса для идентификации геообъектов рядом с точкой

нажатия пользователем кнопки мыши. Мы также рассмотрели использование существующих в GeoDjango форм редактирования, которые позволяют пользователю просматривать и редактировать содержимое геометрий.

Наконец, мы узнали, что нужно делать при добавлении в систему новых геообъектов и как пользоваться шаблоном для подтверждения удаления геообъекта или файла фигур. Мы также научились настраивать внешний вид и функционал встроенных в GeoDjango виджетов редактирования.

На этом наша работа с модулем GeoDjango завершена, а вместе с ним завершается и эта книга. Хотелось бы надеяться, что вы узнали достаточно много о разработке геопространственных приложений и, в частности, о создании геоприложений на Python. Имея этот инструментарий и соответствующую методику в своем распоряжении, теперь у вас есть все, чтобы приступить к разработке своих собственных полнофункциональных геопространственных систем. Желаем успехов!

Глоссарий сокращений и основных терминов

Настоящий глоссарий составлен по материалам глоссария ESRI (http://support.esri.com/sitecore/content/support/Home/other-resources/gis-dictionary/) и веб-сайтов GIS-Lab (http://gis-lab.info/) и ArcGIS (http://desktop.arcgis.com/), OpenStreetMap (http://wiki.openstreetmap.org/) и ряда других.

Сокращения

API (Applications Programming Interface), интерфейс программирования приложений, программный интерфейс.

BGN (US Board on Geographic Names), Совет по географическим названиям США.

BIL (Band Interleaved by Line) с построчным чередованием для систем дистанционного зондирования.

BIP (Band Interleaved by Pixel) с попиксельным чередованием для систем дистанционного зондирования.

BSQ (Band Sequen-tial), формат данных с последовательным чередованием для систем дистанционного зондирования.

CBSA (Core-Based Statistical Area), статистический ареал вокруг городского центра.

CGI (Common Gateway Interface), общий интерфейс шлюза, стандарт интерфейса, используемого для связи внешней программы с веб-сервером.

Coverage, собственный формат данных, используемый в системе ARC/INFO компании ESRI для хранения геообъектов и их топологии.

CSRF (Crosssite Request Forgery), подделка кросс-сайтовых запросов.

CSV (Comma-Separated Values), файл значений, в котором поля разделены запятыми.

DB-API, общий программный интерфейс (API), скрывающий детали реализации каждой отдельной СУБД.

DEM (Digital Elevation Model), цифровая модель высот (ЦМВ).

DISTAL (Distance-based Identification of Shorelines, Towns, and Lakes), образец геоприложения на Python с идентификацией береговых линий, городов и озер в заданном радиусе.

Django, веб-платформа для разработки веб-приложений на Python.

DRG (Digital Raster Graphic), формат цифровой растровой графики для хранения цифровых сканов бумажных карт.

DRG (Digital Raster Graphic), цифровое растровое графическое изображение.

DRM (Digital Relief Model), цифровая модель рельефа (ЦМР).

EGM 96 (Earth Geavitational Model), гравитационная модель Земли 1996.

EPSG (European Petroleum Survey Group), Европейская нефтяная топографическая группа.

ESDI (Earth Science Data Interface), интерфейс с данными наук о Земле (с данными геонауки).

ESRI (Environmental Systems Research Institute), американская компания «Институт исследования систем окружающей среды».

ETM+ (Landsat Enhanced Thematic Mapper Plus), расширенный восьмиканальный мультиспектральный сканирующий радиометр на спутниках Landsat.

GDAL (Geospatial Data Abstraction Library), динамическая библиотека обобщенных моделей геоданных для работы с растровыми данными.

GeoDjango, географическое расширение веб-платформы Django.

GeoJSON, открытый формат для кодирования географических структур данных, который основан на формате для обмена данными JSON.

GEOnet Names Server (GNS), сервер географических названий (СГН) GEOnet.

GEOS, динамическая библиотека на C++ для выполнения операций над двумерными геопространственными геометриями и выполнения их анализа.

GIS (Geospatial Information System), геопространственная информационная система (ГИС).

Git, распределенная система управления версиями ПО.

GitHub, веб-служба для хостинга ИТ-проектов и их совместной разработки.

GLOBE (Global Land One-kilometer Base Elevation), высококачественная глобальная ЦММ с базой в 1 км на пиксел.

GML (Geography Markup Language), язык географической разметки, открытый стандарт на основе XML для обмена данными ГИС.

GNIS (Geographic Names Information System), информационная система географических названий (ИСГН).

GPS (Global Positioning System), система глобального позиционирования.

GSHHG (Global Self-consistent, Hierarchical, High-resolution Geography Database), Глобальная последовательная иерархическая географическая база данных высокого разрешения.

IOGP (The International Association of Oil & Gas Producers), Международная ассоциация производителей нефти и газа.

JSON (JavaScript Object Notation), объектная форма записи JavaScript.

Landsat, проект по непрерывной сборке изображения поверхности планеты Земля на основе спутниковых снимков.

LULC (Land Use and Land Cover), землепользование и ландшафт.

Mac OS X, Unix-подобная операционная система.

Mapnik, динамическая библиотека для конструирования картографических приложений.

MaxMind GeoiP, ПО идентификации местоположения и других характеристик интернет-пользователей для различных областей применения.

NAD 27 (North American Datum), североамериканский датум 1927 года.

NAD 83 (North American Datum), североамериканский датум 1983 года.

Natural Earth, веб-сайт «Естественная планета Земля», предоставляющий векторные и растровые геоданные с высоким, средним и низким разрешениями.

NED (National Elevation Dataset), национальный набор данных рельефа США (НДР).

NoSQL, бессхемные базы данных (без-SQL).

OGC (Open Geospatial Consortium), Открытый геопространственный консорциум, международная организация по стандартизации.

OGR, динамическая библиотека для работы с векторными данными.

OpenLayers, картографическая библиотека на JavaScript.

OSM (OpenStreetMap), некоммерческий веб-картографический проект по созданию подробной карты мира силами участников.

POSIX, (Portable Operating System Interface), переносимый интерфейс операционных систем, набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API).

PostGIS, геопространственное расширение для СУБД PostgreSQL.

PostgreSQL, объектно-реляционная СУБД.

PROJ.4, динамическая библиотека на C для работы с картографическими проекциями.

pyproj, библиотека Python, обертка вокруг динамической библиотеки PROJ.4 на C.

pyscopg2, адаптер СУБД PostgreSQL для Python.

REST (Representational State Transfer), наиболее распространенная форма протоколов обмена сообщениями для взаимодействия компонентов распределенного приложения в сети.

REST API, прикладной программный интерфейс, который определяет набор функций, к которым разработчики могут совершать запросы и получать ответы по протоколу HTTP.

RESTful, архитектура веб-служб на основе технологии REST.

ShapeEditor, геопространственное веб-приложение на Python.

Shapely, библиотека Python, обертка вокруг динамической библиотеки GEOS на C++.

shape-файл, открытая спецификация, разработанная компанией ESRI для хранения и обмена данными ГИС.

shebang («шебанг»), последовательность #! в начале файла скрипта в Unix.

Simple Features, простые геообъекты, стандарт OpenGIS для хранения геоданных и связанных с ними атрибутов.

SRID (Spatial Reference Identifier), идентификатор пространственной привязки.

SQL (Structured Query Language), язык структурированных запросов.

TIGER (Topologically Integrated Geographic Encoding and Referencing System), Система топологически интегрированной географической кодировки и привязки.

TIGER/Line, текстовый формат геоданных Бюро переписи населения США.

TIN (Triangulation Irregular Network), нерегулярная триангуляционная сеть.

TMS (Tile Map Service), Служба сборных цифровых карт.

TSV (Tab-Separated Values), файл значений, в котором поля разделены символом табуляции.

UI (User Interface), пользовательский интерфейс.

URL (Uniform Resource Locator), единообразный определитель местоположения ресурса в сети.

USCB (US Census Bureau), Бюро переписи населения США.

USGS (US Geological Survey), Геологическая служба США.

WGS 84 (World Geodetic System 1984), всемирная система геодезических параметров Земли 1984 года; в отличие от локальных систем, является единой системой для всей планеты.

WKB (Well-known Binary), стандартное двоичное представление, двоичные данные для представления одного геообъекта.

WKT (Well-known Text), стандартное текстовое представление, простой текстовый формат для представления одного геообъекта.

World Borders Dataset, набор данных границ стран мира.

WRS (Worldwide Reference System), глобальная система индексации данных.

WSGI (Web Server Gateway Interface), интерфейс шлюза с веб-сервером, протокол связи веб-сервера с приложением на Python.

ГЛОНАСС, Глобальная навигационная спутниковая система.

ПЗ-90, Параметры Земли 1990 года, датум.

Термины

Аффинное преобразование (affine transformation), или полиномиальное преобразование первого порядка – отображение плоскости в себя, при котором прямые в растровом наборе переходят в прямые в искривленном наборе. Используется для смещения, масштабирования и вращения растрового набора данных, в результате чего квадраты ячеек преобразуются в параллелограммы любого размера и угловой направленности.

Базовая карта, или подложка (basemap) – географическая основа карты в виде растровой карты, которая применяется в ГИС в качестве фона для наложения и нанесения растровых и векторных слоев изображения. Используется для привязки данных, нанесения тематического содержания, ориентирования при работе с картой. Существуют три главных типа подложек – ортотрансформированные аэрофото- и спутниковые снимки и отсканированные карты.

Береговая линия (shoreline) – линия водораздела, или линия соприкосновения поверхности моря или озера с поверхностью суши. В связи с тем, что уровень воды изменяется даже за короткий промежуток времени, береговая линия представляет собой условное понятие, применяемое относительно среднего многолетнего положения уровня водного объекта.

Большой круг (great circle) – сечение шара диаметральной плоскостью, то есть плоскостью, проходящей через центр шара.

Веб-служба (web-service) – это сетевая технология, обеспечивающая межпрограммное (межкомпьютерное) взаимодействие на основе веб-стандартов.

Высота (elevation, height) – расстояние по отвесной линии от уровня отсчета до заданной географической точки. Значение высоты точки называется отметкой.

Географическая карта (map) – изображение модели земной поверхности, содержащее координатную сетку с условными знаками на плоскости в уменьшенном виде.

Географические координаты (geographic coordinates) – угловая мера положения на поверхности Земли, выраженная в градусах широты и долготы.

Географическое положение, или местоположение (location) – положение географического объекта относительно поверхности Земли, а также по отношению к другим объектам, с которыми он находится во взаимодействии, заданное значением координаты.

Геоинформационная система, или геопространственная информационная система (GIS) – информационная система, предназначенная для сбора, хранения, обработки, отображения и распространения данных о пространственно-координированных объектах и явлениях, а также получения на их основе новой информации и знаний. В отличие от других информационных систем, все моделируемые в ГИС объекты и явления имеют пространственную привязку, позволяющую анализировать их во взаимосвязи с другими пространственно определенными объектами; изначально под ГИС подразумевалась «географическая информационная система».

Геокоррекция (georectification) – цифровое выравнивание аэрофото- и спутникового снимка с картой той же области. При геокоррекции и на снимке, и на карте маркируется ряд соответствующих связующих (контрольных) точек, таких как перекрестки улиц. Эти географические положения становятся опорными точками во время последующей обработки снимка.

Геопривязка (georeference) – процесс присвоения значений реальных координат каждой ячейке растра.

Глобальная система индексации данных (Worldwide Reference System, WRS) – глобальная схема разграфки, используемая при каталогизации данных, полученных со спутников Landsat.

Геодезический датум (geodetic datum) – метод смещения и вращения эллипсоида для представления формы Земли, обычно в виде сплюснутого сфероида, который апроксимирует поверхность Земли на локальном или глобальном уровне; является ориентиром для системы геодезических координат.

Геометрия (geometry) – объект структуры данных, нередко рекурсивной по своей природе, который дает геометрическое определение пространственного объекта (геообъекта).

Геообработка (geoprocessing) – управление, обработка и анализ пространственных данных с помощью ГИС.

Динамическая библиотека, или разделяемая библиотека (shared library) – файл с расширением .so, .dll или .dylib, в котором хранится скомпилированный программный код, написанный на C/C++/Fortran и содержащий критические

в плане скорости исполнения процедуры и специализированные функции, которые позволяют расширить производительность и функциональность рабочей программы. Загружается в память по запросу уже работающего процесса, то есть динамически.

Единица измерения карты (map unit) – наземная единица длины, в которой хранятся координаты пространственных данных.

Зона (zone) – это участок земной поверхности, ограниченный двумя меридианами.

Карта мира (world map) – географическая карта, на которой изображен земной шар целиком. Наиболее часто используются политическая, физическая, а также тематические карты мира: тектоническая, климатические, геологическая, почвенная, растительности, зоогеографическая и др.

Картографическое наложение (map overlay) – пространственная операция, в которой две или несколько карт, или слоев, зарегистрированные в единой системе координат, накладываются в цифровой форме либо на прозрачный материал, для того чтобы показать связи между геообъектами, которые занимают одно и то же географическое пространство.

Картографическая (прямоугольная) система координат (Cartesian coordinate system) – прямолинейная система координат со взаимно перпендикулярными осями на плоскости или в пространстве.

Квадрант (quadrant) – в картографической (прямоугольной) системе координат любая из четвертей, образованных центральным пересечением осей X и Y, которая разделяет плоскость на четыре равные части. В евклидовой геометрии измеряемая от центра четверть круга с дугой 90°.

Координата (coordinate) – одна из величин, определяющих положение точки на плоскости или в пространстве.

Линейное кольцо (linear ring) – это замкнутая и одновременно простая ломаная линия. Другими словами, первая и последняя координаты в кольце должны быть одинаковыми, а внутри кольца не должно быть самопересечений.

Лист карты (tileset) – подмножество сегментов, изображающее конкретную карту на конкретном масштабном уровне (уровне приближения).

Ломаная линия (line string) – геометрическая фигура, заданная связанной последовательностью уникальных пар координат x и y; может быть прямой или кривой.

Многоугольник, или полигон (polygon) – площадная геометрическая фигура, заданная связанной последовательностью пар координат x и y, в которой первая и последняя пары координат одинаковые, а все остальные пары – уникальные.

Модель данных (data model) – способ описания однотипных пространственных объектов, включающий способ описания отдельных объектов, топологических отношений между ними, а также дополнительных знаний о всей совокупности объектов в модели.

Мэшап (mashup) – в интернет-картографировании это комбинирование содержимого из нескольких источников данных в одну службу динамического карто-

графирования (dynamic map service); также гибридное приложение, в котором совмещены данные и функциональность из нескольких источников.

Наземные контрольные точки, или точки привязки, реперные точки (ground control points) – точки, используемые для пространственной привязки набора данных в нужную систему координат, представленную этими точками.

Нарезка, или разделение, на регулярные сегменты (tiling) – выделение из крупного набора пространственных данных (обычно растра) управляемого прямоугольного подмножества строк и столбцов согласно заданным границам, как правило, используемых с целью их обработки или анализа без потребления больших объемов памяти компьютера.

Обертка, или обеточный программный код (wrapper) – это промежуточный слой между средой разработки на языке высокого уровня, в данном случае Python, и прикладным программным интерфейсом (API), который предоставляется вебсервисом, динамической библиотекой или СУБД.

Ориентир (landmark) – любой выделяющийся природный или искусственный объект в ландшафте, который используется для определения расстояния, угла направления или географического положения.

Ортодромия (из древнегреческого «прямой путь») – в геометрии кратчайшая линия между двумя точками на поверхности вращения, частный случай геодезической линии.

Ортокоррекция, или ортотрансформирование снимка (orthorectification) – математически строгое преобразование исходного снимка в ортогональную проекцию и устранение искажений, вызванных рельефом, условиями съемки и типом камеры.

Открытый контент (open content) – любое творческое произведение или контент, опубликованный под лицензией, которая явно разрешает копирование и изменение этой информации кем угодно, а не только закрытой организацией, фирмой или частным лицом. Является альтернативной парадигмой использованию копирайта для создания монополий, способствующих целям демократизации знаний.

Пиксел, или ячейка (picture cell) – двумерный (площадной) объект, являющийся элементом регулярной прямоугольной решетки в растровой модели данных; также именуется точками растра.

Привязки Python (bindings) – программный код, благодаря которому программа Python получает доступ к методам и свойствам программного объекта; синонимичен термину *Обретка*.

Пространственный индекс (spatial index) – это индекс базы данных, создаваемый в пространственном столбце таблицы для оптимизации доступа к пространственным данным.

Пространственный объект, или геообъект (feature) – цифровое представление объекта реального мира на карте, включающее координатную привязку (описание геометрии) и набор атрибутов (текстовых и числовых характеристик); также цифровая модель объекта местности.

Прямоугольные (спроецированные) координаты (projected coordinates) – координаты картографической системы координат, полученные в результате математического отображения точки на поверхности Земли на плоскость.

Рельеф (relief) – совокупность неровностей земной поверхности, слагающихся из разнообразных элементарных форм различного порядка.

Сборная карта (tile map) – полная карта всей поверхности Земли или ее части, показывающая конкретное подмножество геообъектов или стилизованная определенным образом.

Светотеневая пластика рельефа (shaded relief) – метод изображения меняющейся высоты рельефа при помощи светотени в зависимости от заданного угла и высоты солнца. Также обозначает растровое изображение с таким методом изображения. Синоним «светотеневая отмывка рельефа» восходит к способу изображения рельефа, при котором постепенное изменение силы тени (или цветного тона) достигается отмывкой кистью или тушевкой карандашом.

Сегмент сборной карты (tile) – единичный фрагмент изображения географической карты четырехугольной формы, представляющий небольшой участок в составе сборной карты.

Сервер сборных цифровых карт (tile map server) – универсальный веб-сервер, в котором реализован протокол службы сборных цифровых карт. На одном сервере сборных цифровых карт может быть размещено несколько служб сборных цифровых карт.

Система координат, или система пространственной привязки (coordinate system) – это метод назначения координат местоположению на поверхности Земли и установления взаимосвязей между наборами таких координат. Система координат позволяет представлять положения в пространстве реального мира при помощи набора координат.

Системный путь среды программирования Python (Python path) – это список каталогов, в которых Python ищет модули, всякий раз исполняя оператор import.

Скользящая карта, или подвижная карта (slippy map) – современная веб-карта с функционалом масштабирования и панорамирования (карта «скользит», когда вы двигаете мышью); также веб-интерфейс для просмотра преобразованных в карту геоданных таким образом.

Слой (layer) – совокупность однотипных пространственных объектов, определенных в одной модели данных на общей территории и в общей системе координат.

Служба сборных цифровых карт (tile map service) – специализированная веб-служба для обеспечения доступа к конкретному подмножеству географических и топографических карт.

Составная точка, или мультиточка (multipoint) – нульмерный объект, который состоит из ненулевого набора несоединенных точек.

Составной многоугольник, или мультиполигон (multipolygon) – двумерный (площадной) объект, который определяется несколькими контурами, заданными в виде последовательности замкнутых непересекающихся линий.

Спецификация простых объектов, или свойств (Simple Features) – стандарт OpenGIS, который определяет хранение географических данных (точка, линия, многоугольник и др.) в цифровом формате с пространственными и непространственными атрибутами.

Статистический ареал вокруг городского центра (core-based statistical area, CBSA) – урбанизированная зона вокруг одного или нескольких крупных городов-ядер с высокой плотностью населения и тесными социально-экономическими связями.

Точка (point) – геометрический элемент, задаваемый парой координат x и y.

Трансформация (transformation) – перевод координат из одной системы координат в другую.

Файл фигур, или шейп-файл, файл форм (shapefile) – векторный формат хранения данных о географическом положении, геометрии и атрибутах географических объектов; содержит один класс геообъектов.

Фигура Земли (the shape of the Earth) – форма земной поверхности. В зависимости от определения фигуры Земли устанавливаются различные системы координат.

Цифровая карта (digital map) – совокупность различных слоев, определенных на общей территории и в общей системе координат; математическая модель изображения цифровой карты.

Цифровая модель местности (ЦММ, digital elevation model, DEM) – ячеистая модель данных, представленная регулярной сеткой высотных отметок в трехмерных координатах (xyz); также формат записи данных высот.

Цифровая модель рельефа (ЦМР, digital relief model, DRM) – часть цифровой модели местности, описывающая форму земной поверхности. ЦМР в геоинформационных системах моделируется с помощью ячеистых моделей данных, называемых обычно DEM и TIN.

Экстент карты (map extent) – предел географического участка, изображаемого на карте, обычно определенного прямоугольником. При динамическом отображении карты на экране компьютера экстент карты может быть изменен путем масштабирования и панорамирования.

Предметный указатель

Д

Е

И

К

Л

М

Н

О

П

Т

У

Ф

Ц

Ш

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным
платежом, выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Эрик Вестра

Разработка геоприложений на языке Python

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Логунов А. В.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 41,8. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

Разработка геоприложений на Python

Написание геопространственных программ предполагает решение таких задач, как группирование данных по географическому положению, хранение и анализ больших массивов информации, выполнение сложных расчетов и построение красочных интерактивных карт. Чтобы делать это хорошо, нужен соответствующий инструментарий и методология, а также полное понимание геопространственных понятий, таких как картографические проекции, геодезические датумы и системы координат.

Эта книга предоставляет обзор главных геопространственных понятий, источников геоданных и наборов инструментов для геообработки. Рассмотрены приемы хранения и доступа к пространственным данным. Показано создание собственного интерфейса со скользящей картой в рамках веб-приложения. Подробно описано создание редактора геоданных на основе географического модуля GeoDjango для веб-платформы Django.

К концу книги вы будете в состоянии уверенно использовать Python для написания собственных геопространственных приложений.

Чему вы научитесь, прочитав эту книгу:

- получать доступ к геоданным, управлять ими и визуализировать из своих программ на Python;
- применять базовые геопространственные понятия, в том числе географическое положение, расстояние, единицы измерения, картографические проекции и геодезические датумы;
- читать и записывать геоданные в векторном и растровом форматах;
- выполнять сложные практические геопространственные расчеты при помощи языка Python;
- хранить геоданные в базе геоданных и получать к ним доступ;
- использовать точки, линии и многоугольники в рамках своих программ на Python;
- преобразовывать геоданные в привлекательные карты при помощи инструментов для геообработки на языке Python;
- конструировать полнофункциональные картографические веб-приложения на основе Python.

Кому адресована эта книга

Эта книга предназначена для опытных разработчиков на языке Python, которые хотели бы освоить концепции геопрограммирования, методы получения и работы с геоданными, решать пространственные задачи и конструировать сложные картографические приложения.

ISBN 978-5-97060-437-3

ДМК ПРЕСС ПОДГОТОВЛЕНО

www.дмк.рф

Интернет-магазин www.dmkpress.com
Книга-почтой: orders@alians-kniga.ru
Оптовая продажа: «Альянс-книга»
(499)782-3889, books@alians-kniga.ru