Интерактивные дашборды и приложения с Plotly и Dash

Элиас Даббас

ДМК ПРЕСС
издательство

Элиас Даббас

Интерактивные дашборды и приложения с Plotly и Dash

Interactive Dashboards and Data Apps with Plotly and Dash

Harness the power of a fully fledged frontend web framework in Python – no JavaScript required

Elias Dabbas

Packt>

BIRMINGHAM – MUMBAI

Интерактивные дашборды и приложения с Plotly и Dash

Используем полноценный веб-фреймворк в Python на всю мощь – без JavaScript

Элиас Даббас

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2023

УДК 004.4
ББК 32.372
Д12

Д12 Элиас Даббас

Интерактивные дашборды и приложения с Plotly и Dash. Используем полноценный веб-фреймворк в Python на всю мощь – без JavaScript / пер. с англ. А. Ю. Гинько. – М.: ДМК Пресс, 2022. – 306 с.: ил.

ISBN 978-5-97060-988-0

Прочитав эту книгу, вы в полной мере освоите фреймворк Dash от Plotly, предоставляющий разработчикам Python блестящие возможности по созданию полноценных интерактивных веб-приложений и дашбордов без знания языка JavaScript.

Вы научитесь создавать различные типы диаграмм; вставлять в приложение разнообразные элементы управления, включая кнопки, выпадающие списки, флажки, календари и т. д. и снабжать приложения динамическими страницами со ссылками. По прочтении книги вы будете обладать необходимыми навыками развертывания полноценных интерактивных приложений и дашбордов, выполнения многоступенчатого рефакторинга кода и оптимизации написанных вами приложений.

Издание адресовано специалистам по работе с данными и аналитикам, желающим больше узнать о своих исходных данных при помощи интерактивных дашбордов.

ISBN 978-1-80056-891-4 (англ.)
ISBN 978-5-97060-988-0 (рус.)

Оглавление

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте **www.dmkpress.com**, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства ДМК Пресс и Packt Publishing очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты **dmkpress@gmail.com**.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Элиас Даббас – практикующий специалист по онлайн-маркетингу, а также обработке и анализу данных. Совместив эти области, он нашел себя в проектировании программного обеспечения с открытым кодом для создания дашбордов и приложений для работы с данными. Также он специализируется на создании программ для интернет-маркетинга.

Элиас является автором популярной библиотеки *advertools* для Python, предлагающей богатый выбор маркетинговых инструментов с уклоном в оптимизацию поисковых систем (SEO), поисковый маркетинг (SEM), сбор данных и текстовый анализ.

О рецензенте

Леонардо Феррейра – бухгалтер, самостоятельно освоивший обработку и анализ данных до уровня Kaggle Grandmaster и выступающий разработчиком платформ в области аналитики данных. Он начал свое обучение в 2017 году и уже через несколько месяцев приступил к работе по изучаемой специальности. С тех пор Леонардо успел поработать в крупных бразильских и международных компаниях, реализовав более сотни проектов с открытым исходным кодом с портфолио на GitHub и Kaggle. Обладает статусом *Top Rated Plus* на фриланссерской платформе *Upwork*, в рамках которой реализовал более 20 проектов по анализу данных. Также интересуется решениями на базе блокчейн-платформы Cardano.

Введение

Фреймворк Dash от Plotly позволяет разработчикам Python создавать полноценные приложения для аналитической работы с данными и интерактивные дашборды. Книга, которую вы держите в руках, призвана помочь вам исследовать богатый функционал фреймворка Dash по визуализации данных и научиться извлекать максимум возможного из исходной информации.

Начнем мы с описания экосистемы Dash, основных пакетов, входящих в состав этого фреймворка, а также сторонних библиотек, позволяющих структурировать данные для вашего приложения.

После этого приступим к созданию первого приложения с использованием фреймворка Dash и добавлению в него базового функционала. Далее вы познакомитесь с такими специфическими элементами приложений, как выпадающий список, флажок, ползунок, календарь и др., а также научитесь связывать их с диаграммами и прочими элементами вывода. В зависимости от данных, которые вы визуализируете, вы будете использовать наиболее подходящие типы диаграмм, включая точечные диаграммы, линейные графики, столбчатые диаграммы, гистограммы, карты и пр., и узнаете, как можно адаптировать их под собственные нужды.

Прочитав эту книгу, вы сможете разрабатывать и развертывать сложные интерактивные дашборды, производить многоступенчатый рефакторинг кода и оптимизировать написанные вами приложения.

Для кого эта книга

Книга, которую вы начинаете читать, предназначена для специалистов по работе с данными и аналитиков, желающих больше узнать о своих исходных данных при помощи интерактивных дашбордов, включающих полный спектр визуализаций. Предполагается, что вы хотя бы на базовом уровне знаете язык программирования Python. Это поможет вам быстрее и лучше усвоить техники, описанные в книге.

Структура книги

Глава 1. Знакомство с экосистемой Dash. В данной главе вы познакомитесь с общей экосистемой фреймворка Dash, пакетами, входящими в его состав, а также сторонними библиотеками. Прочитав эту вводную главу, вы научитесь отличать разные элементы приложения и узнаете, для чего предназначен каждый из них. В качестве бонуса вы даже создадите свое первое простое приложение.

Глава 2. Структура приложений Dash. Из этой главы вы узнаете, как можно добавить созданному ранее приложению интерактивности. Здесь вы познакомитесь с концепцией *обратных вызовов* (callback) и научитесь объединять разные визуальные элементы приложения. Вы также увидите, как

с помощью функций обратного вызова можно позволить пользователю управлять содержимым одного визуального элемента посредством другого.

Глава 3. Работа с объектом Figure. В третьей главе книги вы познакомитесь с ключевым объектом *Figure*, узнаете о его компонентах, а также о способах управления им и преобразовании его в различные форматы. Позже мы используем полученные навыки для создания особых типов диаграмм для нашего приложения.

Глава 4. Подготовка и преобразование данных. Введение в Plotly Express. Здесь вы узнаете о форматах данных, наиболее пригодных для анализа. Также вы познакомитесь с пакетом Plotly Express и увидите, с какой легкостью можно с его помощью создавать диаграммы и связывать данные с элементами визуализации.

Глава 5. Интерактивное сравнение данных при помощи столбчатых диаграмм и выпадающих списков. В этой главе мы немного глубже погрузимся в функционал диаграмм и изучим дополнительные возможности визуализации данных. После этого вы узнаете, как можно дать пользователю возможность выбрать сравниваемые элементы при помощи выпадающих списков.

Глава 6. Исследование переменных при помощи точечной диаграммы и фильтрация наборов данных. В данной главе мы подробно рассмотрим один из наиболее популярных видов визуализации, а именно диаграмму рассеяния или точечную диаграмму. Как и в случае со столбчатой диаграммой, мы рассмотрим различные варианты настройки этого типа визуализации. Точечные диаграммы отличаются очень богатыми возможностями для настройки, включая управление размером точек в зависимости от выбранной переменной, исключение наложения точек друг на друга и вывод диаграммы с большим количеством точек данных.

Глава 7. Работа с географическими картами и обогащение дашбордов при помощи языка разметки Markdown. В этой главе вы познакомитесь с еще одним распространенным типом визуализации. Существует множество способов отображения данных на карте. Мы рассмотрим два наиболее часто используемых: *точечный* (scatter map) и *картограмма* (choropleth map).

Глава 8. Определение частотности данных с помощью гистограмм и построение интерактивных таблиц. Эта глава посвящена разным способам построения гистограмм и их настройки, а также разделению данных различными методами с последующим подсчетом результирующих значений.

Глава 9. Машинное обучение: пусть данные говорят сами за себя. В этой главе вы узнаете о том, как работает кластеризация данных, и научитесь оценивать качество анализа. Также мы рассмотрим технику оценки кластеров и даже разработаем интерактивное приложение с реализацией кластеризации по методу *k*-средних.

Глава 10. Ускорение работы приложений с помощью улучшений функций обратного вызова. Здесь мы поговорим об использовании обратных вызовов на базе шаблонов с целью динамической модификации приложения на основе взаимодействия с пользователем и других факторов.

Глава 11. Ссылки и многостраничные приложения. В данной главе будет представлена новая архитектура, позволяющая создавать многостраничные приложения. Также мы рассмотрим технику использования ссылок в качестве элементов ввода или вывода со взаимодействием с другими элементами приложения.

Глава 12. Развертывание приложения. В этой главе мы обсудим вопросы развертывания созданного приложения на сервере с возможностью доступа к нему пользователям из любой точки мира. Здесь возможны разные варианты, и мы рассмотрим пару простых опций, которые могут оказаться полезными.

Глава 13. Следующие шаги. В заключительной главе книги мы поговорим о том, как можно вывести написанное приложение на новый уровень. Здесь мы дадим определенные рекомендации, советы и ресурсы, которые вам, возможно, захочется изучить самостоятельно.

Как извлечь максимум из книги

Для выполнения примеров из книги вам понадобится стабильное соединение с интернетом.

Если вы читаете книгу в формате PDF, мы рекомендуем вводить программный код вручную или использовать загруженный код из хранилища на GitHub (ссылка будет указана ниже). Это позволит вам избежать ошибок, связанных с копированием и вставкой текста.

Загрузите сопроводительные файлы

Сопроводительные файлы можно загрузить на странице книги на сайте издательства **www.dmkpress.com**.

Загрузите цветные изображения

По следующей ссылке вы можете скачать в виде PDF все рисунки и диаграммы, использованные в книге: **https://static.packt-cdn.com/downloads/9781800568914_ColorImages.pdf**.

Книга в видеофрагментах

Сопроводительные видеофрагменты к этой книге можно посмотреть по адресу **https://bit.ly/3vaXYQJ**.

Условные обозначения

На протяжении книги мы будем использовать следующие условные обозначения и шрифты.

`Код в тексте`: так в тексте книги мы будем обозначать код, имена таблиц баз данных, имена папок, файлов, расширения файлов, пути, ссылки, пользовательский ввод. Пример: «Наш набор данных будет состоять из файлов в папке `data`, находящейся в корне репозитория».

Блоки кода будут выделены следующим образом:

```
import plotly.express as px
gapminder = px.data.gapminder()
gapminder
```

Важные места в коде будут подсвечены жирным шрифтом, как показано ниже:

```
import os
import pandas as pd
pd.options.display.max_columns = None
os.listdir('data')
['PovStats Series .csv',
 'PovStats Country .csv',
 'PovStats Country-Series .csv',
 'PovStats Data .csv',
 'PovStats FootNote .csv']
```

Жирным шрифтом также будут выделяться новые термины, важные слова и текст, который вы видите на экране. Например, таким образом будут обозначаться пункты меню. Пример:

«Еще одним важным столбцом является столбец **Limitations and exceptions**».

> **Советы или важные примечания**
>
> Будут выводиться так.

Часть I

Построение приложения на Dash

В этой вводной части вы познакомитесь с экосистемой фреймворка Dash и напишете свое первое простое приложение с минимальным функционалом.

Содержание этой части:

- глава 1 *«Знакомство с экосистемой Dash»;*
- глава 2 *«Структура приложений Dash»;*
- глава 3 *«Работа с объектом Figure»;*
- глава 4 *«Подготовка и преобразование данных. Введение в Plotly Express».*

Глава 1

Знакомство с экосистемой Dash

При работе с данными происходят постоянные изменения в объеме анализируемых данных, их источниках и типах. В связи с этим очень важно иметь возможность легко и просто комбинировать любые объемы данных из различных источников. Фреймворк *Dash* – это не только про исследование данных. Это про почти все стадии процесса анализа данных: от их поиска до создания полноценной рабочей среды.

В этой вводной главе мы познакомимся с экосистемой Dash и сконцентрируемся на внешнем макете приложения – той его части, с которой взаимодействует пользователь. Прочитав эту главу, вы сможете создать полностью работающее приложение с любыми визуальными элементами, но без интерактивных возможностей.

Темы, которые будут рассмотрены в главе:

- настройка окружения;
- исследование фреймворка Dash и сопутствующих пакетов;
- введение в базовую структуру приложения Dash;
- создание и запуск простого приложения Dash;
- добавление HTML и других компонентов в приложение;
- проектирование макета и управление темами.

Технические требования

В каждой главе будут применяться свои технические требования, но некоторые из них будут актуальны на протяжении всей книги.

Прежде всего у вас должен быть установлен *Python* 3.6 или выше, который можно загрузить по адресу **https://www.python.org**. Также вам понадобится текстовый редактор или любая *интегрированная среда разработки* (integrated development environment – IDE) для написания и редактирования кода.

В этой главе мы будем использовать пакеты Dash, Dash HTML Components и Dash Bootstrap Components, которые можно загрузить вместе с остальными пакетами, следуя инструкции из следующего раздела. Исходный код и данные для этой книги можно скачать в репозитории GitHub по адресу **https://github.com/**

PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash. Как я уже упомянул, в следующем разделе мы детально остановимся на настройке вашего рабочего окружения.

Исходный код к этой главе располагается в хранилище GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_01**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3atXPjc**.

Настройка окружения

Поскольку все пакеты, используемые в этой книге, стремительно развиваются и меняются, вы можете столкнуться с различиями в поведении ваших приложений. Чтобы в точности воспроизвести функционал приложений, заложенный при написании этой книги, мы рекомендуем вам клонировать репозиторий книги, установить версии пакетов, которые использовались при написании книги, и использовать в своих примерах приведенные наборы данных. Откройте командную строку, перейдите в папку, в которой хотите создать проект, и выполните следующие действия.

1. Создайте виртуальное окружение Python в папке с именем `dash_project` (или любой другой на ваше усмотрение). Это также приведет к созданию папки с выбранным именем:

```
python3 -m venv dash_project
```

2. Активируйте виртуальное окружение.

   В Unix или macOS выполните следующую инструкцию:

```
source dash_project/bin/activate
```

   Для Windows инструкция будет такой:

```
dash_project\Scripts\activate.bat
```

3. Перейдите в созданную папку:

```
cd dash_project
```

4. Клонируйте репозиторий книги на GitHub:

```
git clone https://github.com/PacktPublishing/Interactive-
Dashboards-and-Data-Apps-with-Plotly-and-Dash
```

5. В вашей папке должен появиться файл `requirements.txt` с перечислением всех необходимых пакетов и их версий. Вы можете установить все эти пакеты, перейдя в папку репозитория и выполнив команду `install`, как показано ниже:

```
cd Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/
pip install -r requirements.txt
```

В папке `data` вы обнаружите копии наборов данных, которые были загружены с сайта **https://datacatalog.worldbank.org/dataset/poverty-and-equity-database**. Вы всегда можете загрузить свежие версии файлов, но здесь как с версиями пакетов – лучше использовать в точности те данные, которые применялись при написании книги, чтобы получать такие же результаты.

Для корректного отображения объектов и приложений Plotly в JupyterLab необходимо также установить Node.js по адресу **https://nodejs.org**.

Еще вам нужно установить расширение *JupyterLab Plotly*, что можно сделать, запустив следующую инструкцию из командной строки в вашем виртуальном окружении:

```
jupyter labextension install jupyterlab-plotly@4.14.1
```

Обратите внимание, что номер версии в конце строки должен совпадать с версией Plotly, которую вы используете. Вы можете изменить версию, не забыв при этом обновить и сам пакет Plotly.

Теперь вы полностью готовы двигаться дальше. В каждой следующей главе мы будем развивать идеи, озвученные в предыдущих главах, и дорабатывать созданные приложения, улучшая их функционал.

Основная цель – дать вам как можно больше возможностей для практики. Создать отдельный компонент Dash не составляет труда, но при объединении нескольких компонентов в приложении могут начаться сложности. Вы прочувствуете это, когда вам придется обновлять макет приложения и выполнять рефакторинг кода, концентрируясь на деталях, но в то же время не упуская из вида картину в целом.

Итак, окружение мы подготовили, пришло время рассказать о фреймворке Dash.

Исследование фреймворка Dash и сопутствующих пакетов

Хотя в этом нет строгой необходимости, все же вам полезно будет узнать, какие основные компоненты используются в *Dash*, чтобы уверенно чувствовать себя при разработке более сложных приложений и знать, куда обращаться за помощью.

Рис. 1.1. Из чего сделан Dash

Как видно на рис. 1.1, Dash использует фреймворк *Flask* на стороне сервера. Для построения диаграмм применяется графическая библиотека *Plotly* – это не строгое требование, но эта библиотека обладает самыми богатыми возможностями и поддержкой. Библиотека *React* используется для управления компонентами. По сути, любое приложение Dash можно воспринимать как одно-

страничное приложение React. Но гораздо важнее для нас сейчас узнать, какие пакеты используются в процессе создания приложения, и именно об этом мы поговорим далее.

> **Примечание**
>
> Одним из главных преимуществ фреймворка Dash является то, что он позволяет создавать полностью интерактивные приложения и интерфейсы для работы с данными и аналитикой с использованием чистого Python и без необходимости изучать HTML, CSS или JavaScript.

> **Совет**
>
> Тем, кто знаком с библиотекой Matplotlib, будет полезно узнать, что существуют инструменты для преобразования объектов Matplotlib в объекты Plotly. Таким образом, создав элемент в Matplotlib, вы сможете сконвертировать его в Plotly с помощью всего одной функции `mpl_to_plotly`. На момент написания книги такой функционал поддерживался только для Matplotlib версии не выше 3.0.3. Ниже приведен пример использования этой функции.

```
%config InlineBackend.figure_format = 'retina'
import matplotlib.pyplot as plt
from plotly.tools import mpl_to_plotly

mpl_fig, ax = plt.subplots()
ax.scatter(x=[1, 2, 3], y=[23, 12, 34])
plotly_fig = mpl_to_plotly(mpl_fig)
plotly_fig
```

Пакеты, содержащиеся во фреймворке Dash

Dash – это не один большой пакет, который содержит все необходимое. Это, скорее, собрание пакетов, каждый из которых служит конкретной цели. В дополнение, как мы увидим позже, существует большое количество сторонних пакетов, которые используются совместно с Dash, а сообщество создает собственные библиотеки для работы с этим фреймворком.

Ниже приведены основные пакеты, входящие в состав фреймворка Dash, которые мы будем изучать в этой главе:

- **Dash:** это базовый пакет, представляющий основу любого приложения посредством объекта `dash.Dash`. Также в этом пакете представлен функционал для управления интерактивностью и исключениями, что вы увидите при создании приложения;

- **Dash Core Components:** этот пакет содержит интерактивные компоненты, которыми управляет пользователь. Выпадающие списки, календари, ползунки и многое другое – это все есть в этом пакете. В главе 2 мы будем подробно говорить об этих компонентах применительно к интерактивности приложений, а во второй части книги еще более детально обсудим нюансы их использования;
- **Dash HTML Components:** в этом пакете представлены все возможные теги HTML в виде классов Python.Именно здесь происходит преобразование Python в HTML. К примеру, вы можете написать в Python `dash_html_components.H1('Hello, World')`, и этот код сгенерирует следующую разметку HTML: `<h1>Hello, World</h1>` – и соответствующим образом отобразит ее в браузере;
- **Dash Bootstrap Components:** это сторонний пакет, добавляющий фреймворку Dash функциональности Bootstrap. Этот пакет и его компоненты отвечают главным образом за макет и визуальное отображение элементов. Используя его, можно, например, разместить элементы приложения бок о бок или один над другим, определяя их размеры в зависимости от размера окна браузера, или настроить цветовую гамму приложения особым образом.

> **Совет**
>
> Чтобы установить все основные пакеты Dash, достаточно установить главный пакет. Это также позволит сохранить консистентность версий сопутствующих пакетов. Просто выполните инструкцию `pip install dash` из командной строки. Для обновления пакетов воспользуйтесь командой `pip install dash --upgrade`.

Теперь пришло время взглянуть на базовую структуру типичного приложения Dash.

Введение в базовую структуру приложения Dash

На рис. 1.2 условно показан процесс создания приложения Dash. К примеру, у нас есть файл с именем `app.py` (вы можете назвать его по своему усмотрению). Содержимое файла показано в правой колонке рисунка с условным разделением на секции, а в левой приведено описание секции.

Давайте рассмотрим каждый из приведенных этапов отдельно:

- **импорт (стандартная заготовка)**: как и любой модуль Python, мы начинаем написание приложения с импортирования необходимых пакетов, давая им привычные псевдонимы[1];

[1] Здесь и далее: начиная с версии фреймворка Dash 2.0 следует использовать следующий синтаксис импорта: `from dash import html` и `from dash import dcc`, поскольку указанный синтаксис является устаревшим. – *Прим. перев.*

Части приложения	app.py

Импорт (стандартная заготовка)	`import dash`
	`import dash_html_components as html`
	`import dash_core_components as dcc`

Создание экземпляра приложения	`app = dash.Dash(__name__)`

Макет приложения: список HTML и/или интерактивных компонентов	`app.layout = html.Div([`
	`  dcc.Dropdown()`
	`  dcc.Graph()`
	`  ...`
	`])`

Функции обратного вызова	`@app.callback()`
	`  ...`
	`@app.callback()`
	`  ...`

Запуск приложения	`if __name__ == '__main__':` `  app.run_server()`

Рис. 1.2. Структура приложения Dash

- **создание экземпляра приложения**: простой способ создать приложение посредством инициализации переменной `app`. В качестве параметра передается значение `__name__`, чтобы Dash мог легко находить статические ресурсы, используемые в приложении;
- **макет приложения**: этому этапу мы посвятим большую часть данной главы. Именно здесь мы создаем все пользовательские элементы. Для этого мы обычно определяем контейнер (`html.Div`), принимающий в качестве аргумента `children` список дочерних компонентов. Эти компоненты будут последовательно отображаться при запуске приложения – один под другим. В следующем разделе мы создадим простейшее приложение с минималистическим макетом;
- **функции обратного вызова**: эту тему мы подробно начнем обсуждать в главе 2, посвященной интерактивности приложений. Пока вам достаточно будет знать, что здесь определяются функции для связывания визуальных элементов, в результате чего мы получаем функционал приложения. Обычно функции являются независимыми, они не должны быть объявлены внутри контейнера, а их порядок в модуле не имеет значения;
- **запуск приложения**: здесь мы осуществляем так называемый запуск приложения, если применить идиому о запуске модулей Python в качестве скриптов.

Итак, как я и обещал, мы уже готовы к написанию своих первых строк!

Создание и запуск простого приложения Dash

Держа в уме структуру приложения, которую мы только что обсудили, за исключением функций обратного вызова, давайте попробуем построить простейшее приложение.

Создайте файл app.py и введите в него следующий код.

1. Импорт необходимых пакетов с псевдонимами:

```
import dash
import dash_html_components as html
```

2. Создание экземпляра приложения:

```
app = dash.Dash(__name__)
```

3. Создание макета приложения:

```
app.layout = html.Div([
    html.H1('Hello, World!')
])
```

4. Запуск приложения:

```
if __name__ == '__main__':
    app.run_server(debug=True)
```

Позвольте сделать пару замечаний перед запуском приложения. Я настоятельно рекомендую не пользоваться копированием и вставкой кода. Пишите вручную. Вы должны запоминать вводимые конструкции. Кроме того, в процессе написания кода вам будут показываться подсказки в среде выполнения, и очень важно обращать внимание на предлагаемые возможности для компонентов, классов или функций.

Макет нашего приложения содержит единственный элемент, переданный как список элементу html.Div в качестве параметра children. Этот элемент будет преобразован в тег H1.

Заметьте также, что я передал на вход методу app.run_server параметр debug со значением True. Это позволяет активировать инструменты отладки, помогающие при разработке приложения.

Итак, мы готовы запустить наше первое приложение Dash. Для этого из командной строки, находясь в папке с файлом app.py, выполните следующую инструкцию:

```
python app.py
```

Если третья версия Python не установлена в вашей системе по умолчанию, вам может потребоваться указать номер версии вручную, как показано ниже:

```
python3 app.py
```

Вы увидите вывод, показанный на рис. 1.3, из которого следует, что приложение запущено.

```
> python app.py
Dash is running on http://127.0.0.1:8050/

 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
```

Рис. 1.3. Вывод командной строки после запуска приложения

Поздравляем! Ваше первое приложение Dash готово! Теперь, если вы введете в адресную строку браузера ссылку, указанную в выводе командной строки (**http://127.0.0.1:8050**), то увидите строку *Hello, World!* в виде заголовка первого уровня (H1). Кроме того, в выводе командной строки указано, что в данный момент запущено серверное приложение Flask с именем `app`, а также присутствует предупреждение о том, что это сервер для разработки, а не для выпуска приложения. О развертывании приложений мы будем говорить позже, сейчас же вам достаточно знать, что этого сервера нам вполне хватит для разработки и тестирования приложения. Вывод приложения показан на рис. 1.4.

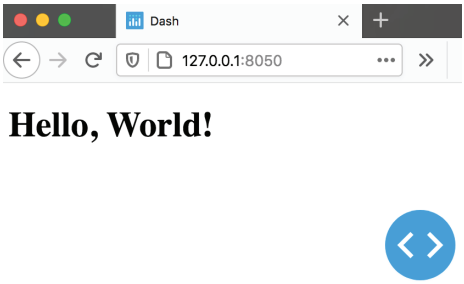


Рис. 1.4. Приложение в браузере

Под текстом, выведенным в виде заголовка первого уровня, вы можете заметить голубой кружок со стрелками. Щелкните по нему, и откроются дополнительные опции *отладки*, которые пригодятся вам при наличии в приложении функций обратного вызова и появлении ошибок. Если в качестве параметра при запуске приложения передать `debug=False`, эта кнопка отображаться не будет.

Теперь, когда вы создали простое приложение Dash и разобрались в базовой структуре макета, пришло время присмотреться к двум пакетам, которые используются для добавления и управления визуальными элементами. Первый пакет именуется **Dash HTML Components**, а второй – **Dash Bootstrap Components**.

Добавление HTML и других компонентов в приложение

С этого момента и до конца главы мы будем главным образом говорить об атрибуте нашего приложения `app.layout` и вносить в него изменения. Делать

это очень легко – достаточно просто добавлять элементы в список, поступающий на вход элемента `html.Div` в качестве параметра `children`:

```
html.Div(children=[component_1, component_2, component_3, ...])
```

Добавление компонентов HTML в приложение Dash

Поскольку доступные компоненты в пакете в точности соотносятся с соответствующими тегами HTML, этот пакет можно назвать достаточно устойчивым и стабильным. Давайте рассмотрим параметры, общие для всех его компонентов.

На момент написания книги **Dash HTML Components** включал в себя 131 компонент, общими для которых являются 20 параметров. Поговорим о самых популярных из них, которые мы часто будем использовать в этой книге:

- `children`: это основной (и первый) контейнер, вмещающий содержимое компонента. В нем может размещаться как один элемент, так и список элементов;
- `className`: то же, что и атрибут `class`, но под другим именем;
- `id`: несмотря на то что в данной главе мы не будем использовать этот параметр, он играет важнейшую роль при добавлении элементам интерактивности. В дальнейшем при построении приложения мы будем очень часто обращаться к этому свойству. Ну а пока вам достаточно знать, что вы можете присваивать своим компонентам произвольные идентификаторы, по которым впоследствии сможете обращаться к ним при настройке интерактивности;
- `style`: похож по назначению на одноименный атрибут HTML, но имеет некоторые отличия. Во-первых, атрибуты в нем задаются в так называемом верблюжьем стиле (camelCase). Допустим, вам нужно задать показанные ниже атрибуты:

```
<h1 style="color:blue; font-size: 40px; margin-left:
20%">A Blue Heading</h1>
```

Для этого вам придется определить их следующим образом:

```
html.H1(children='A Blue Heading',
    style={'color': 'blue',
        'fontSize': '40px',
        'marginLeft': '20%'})
```

Вы наверняка заметили, что значение атрибуту `style` присваивается с использованием словаря Python.

Другие параметры подчиняются собственным законам и правилам в зависимости от компонента, которому они принадлежат. Давайте потренируемся в добавлении элементов HTML в наше приложение. Вернемся к нашему исходному файлу `app.py`, поэкспериментируем с другими элементами HTML и

снова попробуем запустить приложение. Начало и конец приложения оставьте прежними, а изменить нужно главным образом `app.layout`, как показано ниже:

```
...
app = dash.Dash(__name__)
app.layout = html.Div([
    html.H1('Poverty And Equity Database',
        style={'color': 'blue',
            'fontSize': '40px'}),
    html.H2('The World Bank'),
    html.P('Key Facts:'),
    html.Ul([
        html.Li('Number of Economies: 170'),
        html.Li('Temporal Coverage: 1974 - 2019'),
        html.Li('Update Frequency: Quarterly'),
        html.Li('Last Updated: March 18, 2020'),
        html.Li([
            'Source: ',
            html.A('https://datacatalog.worldbank.org/dataset/poverty-and-
equity-database', href='https://datacatalog.worldbank.org/dataset/poverty-
and-equity-database')
        ])
    ])
])
...
```

В результате наше приложение приобретет вид, показанный на рис. 1.5.

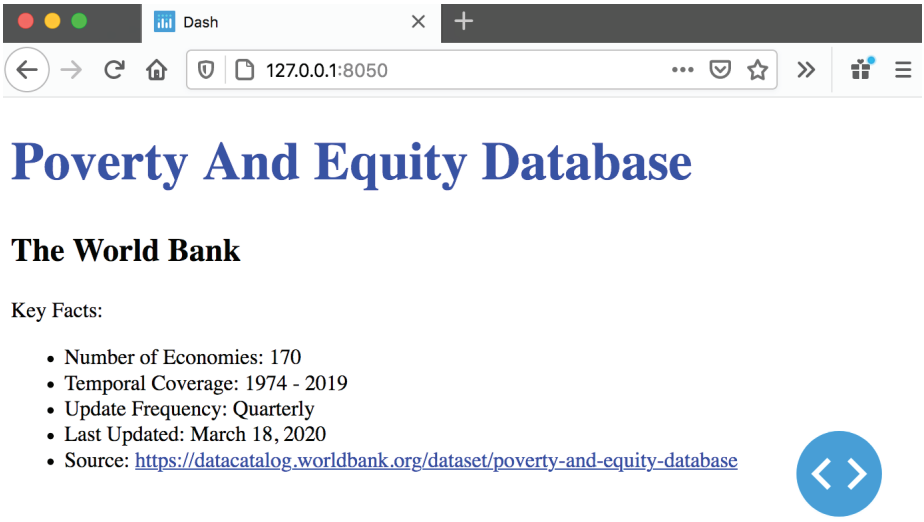


Рис. 1.5. Обновленное приложение, запущенное в браузере

> **Совет**
>
> Если вы знакомы с тегами языка разметки HTML, у вас не будет никаких проблем. Если нет, советую отдельно ознакомиться с этим языком. Начать можно на сайте W3Schools по адресу **https://www.w3schools.com/html**.

В обновлении мы лишь добавили элемент `<p>` и неупорядоченный список `<ul>` с несколькими элементами `<li>` (с использованием списка в Python), последний из которых содержит ссылку в виде элемента `<a>`.

Обратите внимание, что все эти компоненты реализованы в Python в виде классов, в связи с чем их имена начинаются с заглавной буквы, как предписано соглашением об именовании объектов: `html.P`, `html.Ul`, `html.Li`, `html.A` и т. д.

Поэкспериментируйте самостоятельно с другими элементами и атрибутами HTML.

Проектирование макета и управление темами

Мы обсудили базовую структуру приложения Dash и рассмотрели ее основные элементы: импорт пакетов, создание экземпляра и макета приложения, функции обратного вызова (о них мы будем подробно говорить в следующей главе), а также запуск приложения. Мы также создали простейшее приложение и добавили в него несколько элементов HTML. Теперь мы готовы вывести наше приложение на новый уровень с точки зрения оформления макета. Давайте продолжим работать с атрибутом `app.layout`, но для более мощного и гибкого управления им привлечем пакет **Dash Bootstrap Components**.

Bootstrap представляет собой набор инструментов, облегчающих настройку макетов веб-страниц. Ниже мы перечислим основные преимущества и удобства, приобретаемые при использовании этого компонента:

- **темы**: как вы совсем скоро увидите, изменить тему приложения можно с помощью всего одного дополнительного аргумента при его создании. В пакете **Dash Bootstrap Components** содержится целый набор тем, которые вы можете менять и выбирать;
- **координатная сетка**: Bootstrap позволяет удобно размещать элементы приложения в соответствии с сеточным макетом с колонками и строками и не заботиться о пикселях и процентах. При этом тонкая настройка макета вам тоже останется доступна, и вы сможете воспользоваться ей при необходимости;
- **чувствительность и динамичность**: при наличии огромного множества возможных размеров экрана очень трудно бывает настроить макет приложения так, чтобы он нормально выглядел на всех устройствах. Bootstrap решает эту проблему за нас, при этом вы также можете сами управлять тем, как будут меняться размеры элементов при изменении размера экрана;
- **встроенные компоненты**: Bootstrap предлагает большое число удобных компонентов интерфейса, которыми можно при желании восполь-

зоваться. Кнопки, выпадающие меню, вкладки – это лишь малая часть того, что предлагает пакет;

- **кодировка цветов**: также в вашем распоряжении будет набор кодированных цветов, которые можно использовать при необходимости вывести ошибку, предупреждение или просто информационное сообщение для пользователя.

Давайте рассмотрим эти пункты по отдельности.

Темы

Посмотрим, как просто можно изменить тему приложения. Добавьте в наш файл `app.py` строку с импортом пакета и дополнительный аргумент в функцию создания экземпляра приложения, как показано ниже:

```
import dash_bootstrap_components as dbc
…
app = dash.Dash(__name__, external_stylesheets=[dbc.themes.BOOOTSTRAP])
…
```

Снова запустив приложение, вы увидите, что его оформление изменилось. На рис. 1.6 представлены другие темы с указанием в нижней части страниц их названий.

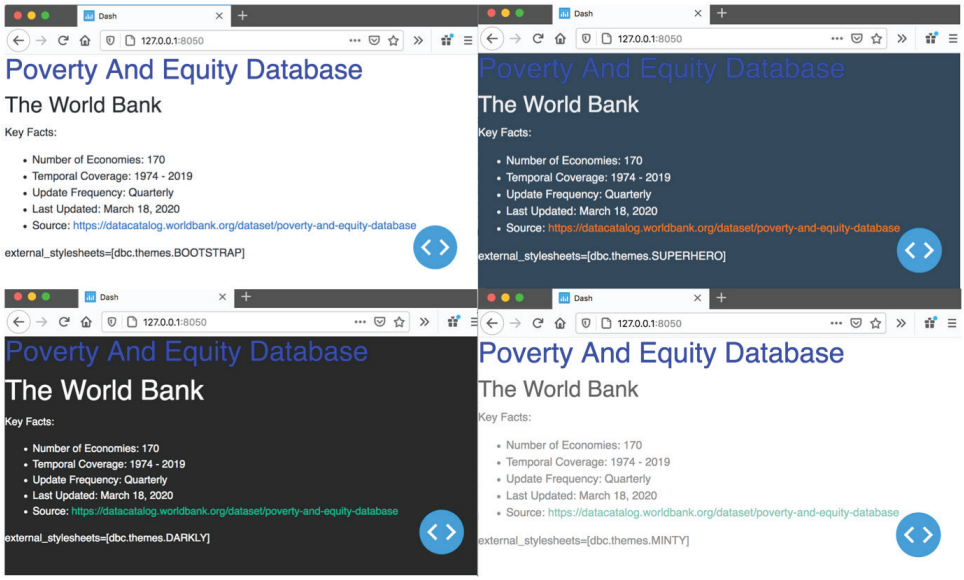


Рис. 1.6. Установка темы приложения

Как видите, достаточно одного аргумента, чтобы полностью изменить внешний вид и восприятие приложения. Также заметьте, что в элементе `<h1>` мы вручную переопределили цвет и размер шрифта при помощи аргумента `style`. Для цвета текста мы задали значение `"blue"`, а для размера – `"40px"`. Обычно

так делать не следует. Обратите внимание, как плохо читается синий текст на темных темах. Так что будьте с этим очень осторожны.

Координатная сетка и чувствительность к изменениям

Еще одно преимущество, которое дает Bootstrap, состоит в возможности использовать в приложении координатную сетку. Мы уже добавляли в элемент `html.Div` список дочерних визуальных элементов с помощью параметра `children`. В этом случае каждый элемент занимает всю доступную ширину окна, а по высоте использует минимальное место, позволяющее отобразить свое содержимое. Порядок элементов в переданном списке строго определяет их расположение на экране.

Размещение элементов в колонках

Установка всех параметров элементов посредством атрибута `style` – занятие весьма трудоемкое, а результат может оказаться не таким, как мы себе представляли. Вам необходимо учитывать слишком много факторов, и в какой-то момент все может пойти не так. С Bootstrap вам достаточно определить колонку, которая будет существовать как отдельный независимый экран, отображая размещенные в ней элементы друг под другом. При этом каждый элемент будет занимать всю доступную ширину этого «экранчика». Что касается ширины колонок-контейнеров, ее можно задать довольно гибко. Координатная сетка условно делит всю ширину экрана на 12 равных долей, и при определении ширины колонок вы можете задавать значение от 1 до 12 включительно. На рис. 1.7 показано, как могут быть определены колонки и как они могут отображаться на экранах разных размеров.

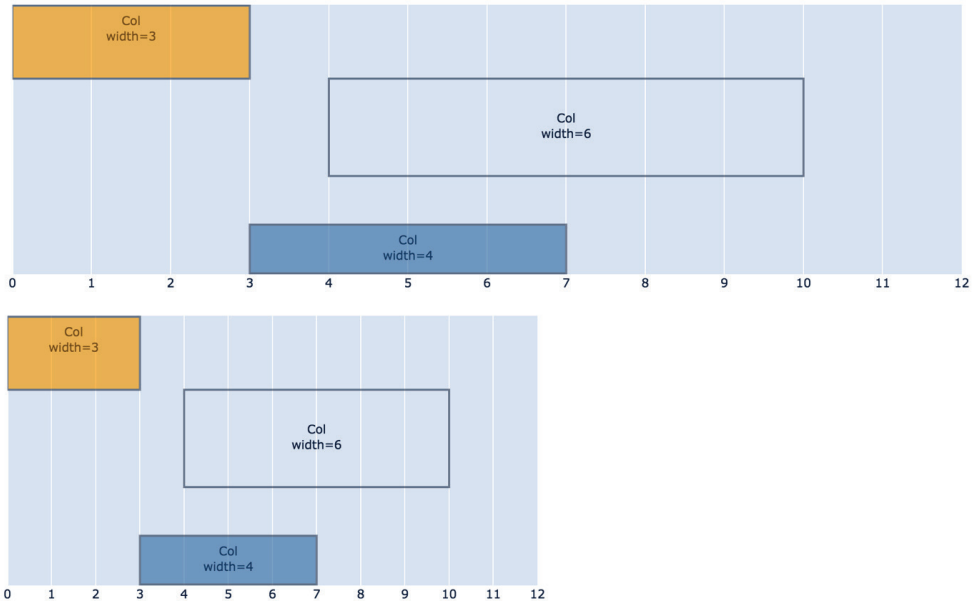


Рис. 1.7. Один и тот же колоночный макет на двух разных экранах

Как видите, размещение элементов на экране оказалось идентичным, а изменение их размеров произошло автоматически с сохранением исходных пропорций.

Но такая схема размещения элементов может не всегда вам подходить. При сильном сужении экрана может быть уместнее расширять колонки, чтобы их содержимое оставалось легкочитаемым. Это можно сделать, указав ширину колонок для пяти *вариантов ширины экрана*: `xs` (очень маленькая), `sm` (маленькая), `md` (средняя), `lg` (большая) и `xl` (очень большая). Именно так именуются и параметры, которые вы можете задать для каждой колонки с элементами.

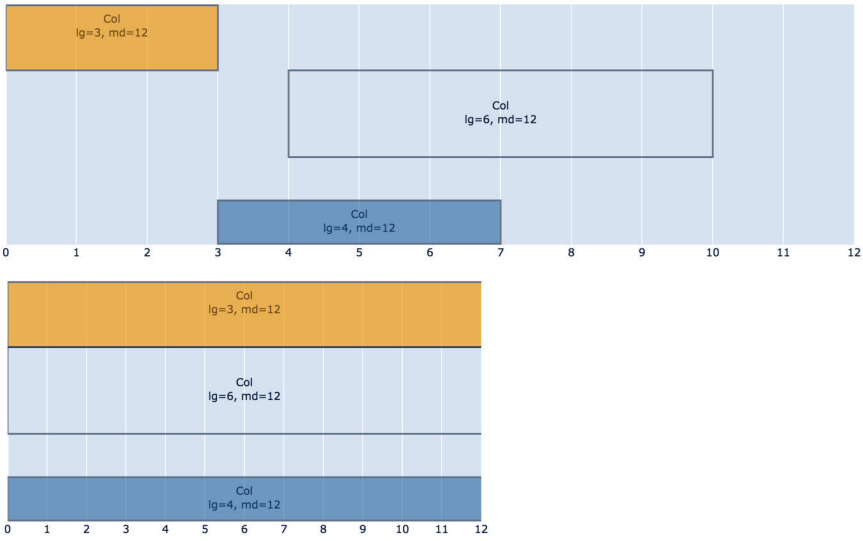


Рис. 1.8. Контроль ширины колонок в зависимости от размера экрана

На рис. 1.8 наглядно видно, как можно это реализовать путем указания двух аргументов для каждой колонки. Установить эти аргументы проще простого, и это показано ниже:

```
import dash_boostrap_components as dbc
dbc.Col(children=[child1, child2, ...], lg=6, md=12)
```

Инструкция `lg=6, md=12` означает наше желание видеть эту колонку с шириной 6 на большом экране (`lg`), что эквивалентно половине ширины экрана. На экранах со средней шириной (`md`) мы бы хотели, чтобы колонка занимала всю доступную ширину окна, для чего и установили соответствующему параметру значение 12.

Возможно, вас интересует, как нам удалось подвесить элементы на произвольном удалении от боковых границ экрана. Дело в том, что параметры размеров можно задавать при помощи словарей, одним из ключей в которых может быть **`offset`**, позволяющий указать смещение колонки от левой границы экрана, как показано ниже:

```
dbc.Col(children=[child1, child2, ...], lg={'size': 6, 'offset': 4}, md=12)
```

Как видите, параметр `lg` компонента `Col` принимает на вход словарь, в котором мы явно указали, что колонка должна размещаться с отступом в четыре позиции от левой границы.

Наконец, если вы хотите разместить колонки бок о бок, вам необходимо поместить их в одну строку (`Row`), как показано на рис. 1.9.

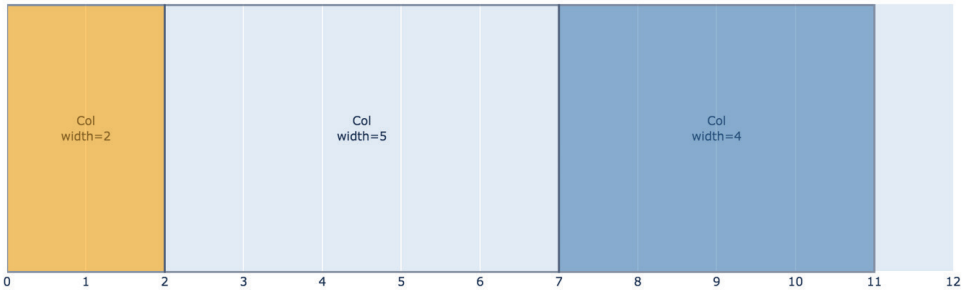


Рис. 1.9. Колонки, размещенные встык по горизонтали

Чтобы собрать такой макет, достаточно поместить все наши колонки в список и подать его на вход элементу `Row` в качестве параметра `children`, как показано ниже:

```
dbc.Row([
    dbc.Col('Column 1', width=2),
    dbc.Col('Column 2', width=5),
    dbc.Col('Column 3', width=4),
])
```

Встроенные компоненты

Конечно, мы не сможем показать в деле все встроенные компоненты Bootstrap, но некоторые из них обязательно продемонстрируем, тем более что их очень просто создавать и добавлять в приложение. Документацию по всем компонентам библиотеки можно найти по адресу **https://dash-bootstrap-components.opensource.faculty.ai**. Скоро мы покажем на примере пару встроенных компонентов.

Кодировка цветов

В своем приложении вы можете использовать любую палитру цветов на ваше усмотрение, но Bootstrap предлагает набор именованных цветов, основанных на информации, которую вы пытаетесь донести до пользователя. В некоторых компонентах вы можете установить один из таких цветов при помощи параметра `color`, и пользователю будет понятно, что вы хотите сказать. К примеру, если установить для определенного компонента параметр `color="danger"`, он станет красным, а если `color="warning"`, то желтым. Список предустановленных цветов содержит следующие значения: `primary`, `secondary`, `success`, `warning`, `danger`, `info`, `light` и `dark`.

Добавление компонентов Dash Bootstrap в приложение

Давайте добавим в наше приложение связанные компоненты `Tabs` и `Tab`. Наверное, вы уже догадались по названиям, что компонент `Tabs` является не чем иным, как контейнером для `Tab`. Все, что нам нужно, – это добавить какую-то информацию на страницы и организовать их в виде вкладок, как показано на рис. 1.10.

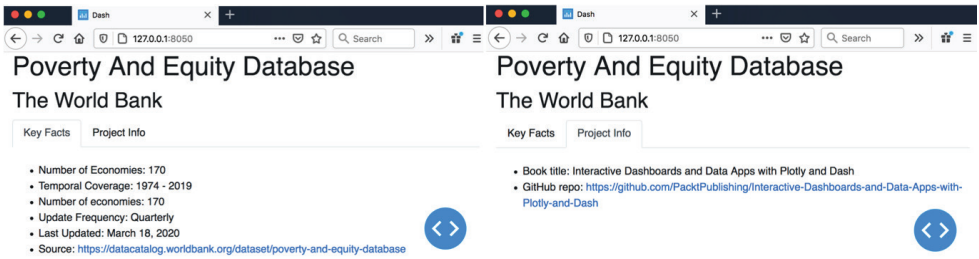


Рис. 1.10. Добавление вкладок в приложение

> **Совет**
>
> Одним из важнейших навыков при изучении фреймворка Dash является умение выполнять *рефакторинг* кода. Хотя наше приложение еще довольно простое, очень важно обладать всеми необходимыми знаниями, для того чтобы постоянно дорабатывать его без потери уже реализованного функционала. И чем больше компонентов будет в вашем приложении, тем более важную роль будет играть ваше умение держать все в голове. Я всегда советую дорабатывать приложение вручную, а не копировать код – так вы быстрее овладеете навыками рефакторинга.

Чтобы создать вкладки и наполнить их контентом, показанным на рис. 1.10, необходимо внести в наше приложение следующие изменения:

```
html.H2('The World Bank'),
dbc.Tabs([
    dbc.Tab([
        html.Ul([
            # тот же код с добавлением неупорядоченного списка
        ]),

    ], label='Key Facts'),
    dbc.Tab([
        html.Ul([
            html.Br(),
            html.Li('Book title: Interactive Dashboards and Data Apps with
Plotly and Dash'),
```

```
            html.Li(['GitHub repo: ', html.A('https://github.com/
PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash',
href='https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-
Apps-with-Plotly-and-Dash')])
    ])
], label='Project Info')
```

Полный код приложения должен выглядеть так:

```
import dash
import dash_html_components as html
import dash_bootstrap_components as dbc

app = dash.Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])

app.layout = html.Div([
    html.H1('Poverty And Equity Database',
            style={'color': 'blue',
                   'fontSize': '40px'}),
    html.H2('The World Bank'),
    dbc.Tabs([
       dbc.Tab([
           html.Ul([
               html.Br(),
               html.Li('Number of Economies: 170'),
               html.Li('Temporal Coverage: 1974 - 2019'),
               html.Li('Update Frequency: Quarterly'),
               html.Li('Last Updated: March 18, 2020'),
               html.Li([
                   'Source: ',
                   html.A('https://datacatalog.worldbank.org/dataset/
poverty-and-equity-database',
                          href='https://datacatalog.worldbank.org/dataset/
poverty-and-equity-database')
               ])
           ])

       ], label='Key Facts'),
        dbc.Tab([
           html.Ul([
               html.Br(),
               html.Li('Book title: Interactive Dashboards and Data Apps
with Plotly and Dash'),
```

```
                html.Li(['GitHub repo: ',
                         html.A('https://github.com/PacktPublishing/
Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash',
                                href='https://github.com/PacktPublishing/
Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash')
                         ])
            ])
        ], label='Project Info')
    ]),
])


if __name__ == '__main__':
    app.run_server(debug=True)
```

Как видите, мы добавили в приложение один элемент `Tabs`, в котором разместили два элемента `Tab`. В первом из них мы оставили написанный ранее код с неупорядоченным списком, а во втором также поместили список с другим содержимым. Ну, ладно, это содержимое вы можете и скопировать! Также обратите внимание, как задаются названия вкладок при помощи параметра `label`.

Теперь вы можете запустить обновленное приложение и убедиться, что вкладки отображаются так, как вы и предполагали[2].

Что ж, пришло время добавить нашему приложению интерактивности!

Заключение

Основным в этой главе было то, что вы научились создавать приложения Dash и даже успели понять, насколько это на самом деле просто. Также мы обсудили содержимое базовых пакетов фреймворка Dash, служащих для добавления визуальных элементов в приложение. Вы приобрели достаточно знаний, чтобы создавать даже сложные макеты с размещением на них любого числа элементов. При этом мы пока не делали ничего сложного и далее в этой книге продолжим обсуждать эти и другие компоненты, чтобы вы могли набить руку и научиться лучше обращаться с ними.

В следующей главе мы попробуем оживить добавленные в приложение элементы, придав им интерактивности. Мы изменим приложение таким образом, чтобы пользователь мог сам делать выбор и анализировать данные в том разрезе, в котором ему необходимо.

[2] Приложение проверено на версиях фреймворка Dash 1.19.0 и актуальной на момент перевода книги – 2.4.1. – *Прим. перев.*

Глава 2

Структура приложений Dash

Итак, вы уже готовы к знакомству с механизмами, обеспечивающими интерактивность приложений Dash. Освоив функции обратного вызова, позволяющие связывать разные элементы приложения, и вооружившись тем, что узнали из первой главы книги, вы сможете с легкостью превращать простые наборы данных в интерактивные приложения Dash с минимальными усилиями. Оставшаяся часть книги будет посвящена тому, как сделать приложения еще лучше и расширить их функционал. Но первых двух глав будет вполне достаточно для создания визуальных макетов приложений и добавления интерактивности. Эта глава будет главным образом посвящена функциям обратного вызова.

Темы, которые будут рассмотрены:

- использование Jupyter Notebook для запуска приложений Dash;
- создание чистой функции на Python;
- знакомство с параметром ID компонентов Dash;
- использование элементов ввода и вывода;
- внедрение функции в приложение;
- запуск вашего первого интерактивного приложения.

Технические требования

В дополнение к пакетам, которые мы использовали в первой главе (например, Dash, Dash HTML Components и Dash Bootstrap Components), мы будем работать с очень важным пакетом Dash Core Components. Также мы научимся запускать приложения Dash в окружении Jupyter Notebook, для чего будем использовать пакет `jupyter_dash` вместе с JupyterLab. Позже в этой главе, когда мы внедрим в приложение новый функционал, мы применим пакет ~~pandas~~ для манипулирования данными.

Код из этой главы можно найти на GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_02**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3tC0ZsW**.

Использование Jupyter Notebook для запуска приложений Dash

Чтобы запустить приложение Dash в среде Jupyter Notebook, необходимо сделать минимальные изменения в коде импорта и создания экземпляра приложения. Здесь нам пригодится пакет `jupyter_dash`. По сути, вся разница сводится к тому, что в момент инициализации приложения мы будем создавать объект **JupyterDash** вместо Dash, как показано ниже:

```
from jupyter_dash import JupyterDash
app = JupyterDash(__name__)
```

Одним из преимуществ запуска приложений в окружении Jupyter Notebook является то, что в нем легче вносить мелкие изменения в приложение, повторно запускать его и сразу видеть результат. Работая с внешней средой разработки (IDE), командной строкой и браузером, вам приходится все время переключаться между окнами, тогда как в Jupyter Notebook вы все делаете в одном окне. Это значительно облегчает разработку и тестирование приложения.

Пакет `jupyter_dash` также предоставляет вам выбор режима запуска приложения. Вы вольны выбрать один из следующих трех режимов:

- `external`: приложение будет запускаться в отдельном окне браузера – так, как мы делали до сих пор;
- `inline`: запуск приложения будет происходить прямо в ноутбуке – непосредственно под ячейкой с кодом;
- `jupyterlab`: приложение будет открываться в отдельной вкладке в JupyterLab.

Также вы можете указать желаемую ширину и высоту приложения, как показано ниже:

```
app.run_server(mode='inline', height=600, width='80%')
```

Как видите, размеры приложения можно задавать как в абсолютных величинах – в пикселях, так и в относительных – в процентах от размера экрана. Во втором случае значение размерности должно быть строковым.

Но есть еще одно важное преимущество использования Jupyter Notebook для запуска приложений, и это…

Изоляция функционала для упрощения процесса разработки и отладки

При разработке и запуске приложений вы неизбежно сталкиваетесь с возникновением ошибок. Чтобы лучше отлавливать их, необходимо выделить причину и создать максимально простой воспроизводимый пример, в котором эти ошибки будут возникать. Только после этого вы можете приступить к решению проблемы и просить помощи у сообщества. Никто не хочет долго ждать, перед тем как возникнет ошибка. Таким образом, мы можем изолиро-

вать все новые возможности приложения перед их внедрением в финальный продукт, чтобы лучше проверить их.

Так что все новинки лучше сначала реализовывать в изолированном окружении с созданием простейшего приложения, включающего только этот функционал. И только после того как убедитесь, что с новым кодом все в порядке, можно сохранить копию приложения для надежности и внедрять в него новинки. Это поможет нам также при желании откатиться в будущем к определенному этапу разработки приложения.

Итак, начнем с выпадающего списка с тремя элементами для выбора. Пользователь приложения должен будет выбрать один из вариантов и увидеть под списком имя элемента, на котором он остановился. На рис. 2.1 показан внешний вид выпадающего списка.

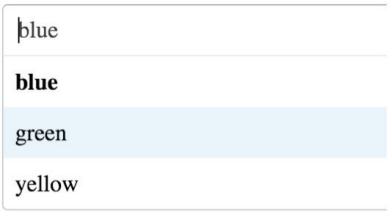


Рис. 2.1. Выбор пользователя на основании активного элемента в списке

Приведенный ниже код позволит создать выпадающий список, но без отображения выбранного пользователем элемента.

1. Импортируйте необходимые пакеты с их псевдонимами:

```
from jupyter_dash import JupyterDash
import dash_core_components as dcc
import dash_html_components as html
```

2. Создайте экземпляр приложения:

```
app = JupyterDash(__name__)
```

3. Создайте макет приложения. Здесь нам понадобится новый объект Dash Core Components **Dropdown**. Подробнее о нем мы поговорим позже, а пока просто используем его атрибут `options` для перечисления вариантов, из которых сможет выбирать пользователь. Этот атрибут инициализируется при помощи списка словарей – по одному для каждого элемента, в которых ключ `label` указывает на то, как пользователь будет видеть элемент в списке, а ключ `value` – на само внутреннее значение:

```
app.layout = html.Div([
   dcc.Dropdown(options=[{'label': color, 'value': color}
```

```
                                for color in ['blue', 'green', 'yellow']]),
    html.Div()
])
```

4. Запустите приложение как обычно, но с небольшим изменением – укажите для него режим `inline` для облегчения интерактивной работы в JupyterLab:

```
if __name__ == '__main__':
    app.run_server(mode='inline')
```

На рис. 2.2 показано, как код должен выглядеть в окружении ноутбука.

```
1  from jupyter_dash import JupyterDash
2  import dash_core_components as dcc
3  import dash_html_components as html
4
5  app = JupyterDash(__name__)
6
7  app.layout = html.Div([
8      dcc.Dropdown(options=[{'label': color, 'value': color}
9                            for color in ['blue', 'green', 'yellow']]),
10     html.Div()
11
12 ])
13 if __name__ == '__main__':
14     app.run_server(mode='inline')
```

Рис. 2.2. Приложение Dash, запущенное в JupyterLab

Уверен, вы заметили пустой элемент `html.Div`, добавленный под выпадающим списком. Давайте разберемся, как вписать блок с выбором пользователя в структуру приложения и реализовать оставшийся функционал. Создадим функцию для связки списка с пустым тегом `div`.

Создание чистой функции на Python

Мы будем использовать *функцию* для извлечения выбранного элемента из выпадающего списка, его обработки и возврата значения, которое поможет оповестить пользователя о сделанном выборе.

Наша функция будет настолько простой, что не потребует дополнительного описания:

```
def display_selected_color(color):
    if color is None:
```

```
        color = 'nothing'
    return 'You selected ' + color
```

Если пользователь не выберет ни одного элемента или отменит ранее сделанный выбор в списке, то переменной `color` будет установлено значение `'nothing'`, и функция вернет информацию о том, что выбор сделан не был. В противном случае будет возвращен текст с выбранным цветом. Далее в этой главе мы напишем более сложную функцию для получения информации о странах.

Функция по своей сути является процедурой. Она принимает на вход один или несколько параметров, обрабатывает их и возвращает одно или несколько значений. Что для нашей функции может быть *элементом ввода* (`Input`) и что должно происходить с ее *элементом вывода* (`Output`)? Давайте ответим на эти вопросы, размышляя в рамках компонентов из нашего макета.

Очевидно, что элементом ввода для функции должен быть выпадающий список. Тогда после обработки его значений должен быть возвращен элемент вывода, который будет непосредственно влиять на содержимое элемента `html.Div` под списком. На рис. 2.3 показано, чего мы хотим добиться. Нам нужно каким-то образом связать элемент ввода (выпадающий список) с элементом вывода (тег `div`), и сделаем мы это при помощи нашей функции-посредника.

Рис. 2.3. Элементы ввода/вывода и функция

Чтобы этот план сработал в рамках приложения, нам необходимо как-то дать понять функции, что для нее будет являться вводом, а что выводом.

А для этого нужно иметь возможность обращаться к элементам по идентификатору...

Знакомство с параметром ID компонентов Dash

Как мы уже вскользь упоминали в главе 1, каждый компонент Dash обладает параметром *id*, позволяющим уникально его идентифицировать. Обеспечение уникальности элементов на макете – основная задача этого параметра.

> **Примечание**
>
> Существуют более продвинутые техники использования параметра `id`, о которых мы поговорим в поздних главах книги. На данном этапе вам будет достаточно того, что этот параметр служит для идентификации элементов.

С развитием приложения растет и потребность понятного и описательного именования его элементов. Если элемент не обладает интерактивностью, параметр `id` для него можно не указывать. В противном случае без него будет просто не обойтись. В приведенном ниже фрагменте кода показано, как можно задать параметр `id` для элементов макета:

```
html.Div([
    html.Div(id='empty_space'),
    html.H2(id='h2_text'),
    dcc.Slider(id='slider'),
])
```

Теперь давайте зададим описательные идентификаторы для выпадающего списка и элемента `div` в нашем изолированном приложении:

```
app.layout = html.Div([
    dcc.Dropdown(id='color_dropdown',
                 options=[{'label': color, 'value': color}
                     for color in ['blue', 'green', 'yellow']]),
    html.Div(id='color_output')
])
```

С точки зрения макета наше приложение завершено. От приложения из главы 1 оно отличается тем, что мы задали для визуальных элементов параметры `id` и запустили приложение в окружении Jupyter Notebook. Теперь, когда мы можем уникально идентифицировать наши элементы, можно определить, какие из них будут являться *элементами ввода* (`Input`), а какие – *элементами вывода* (`Output`). На рис. 2.4 показана слегка измененная схема нашего приложения с указанием идентификаторов элементов, которые должны стать вводом и выводом для функции.

Теперь мы можем привязать наши элементы к функции при помощи этих идентификаторов.

Использование элементов ввода и вывода

На следующем шаге мы должны определить и установить, какие компоненты приложения должны стать элементами ввода функции, а какие – вернуться в виде вывода для отображения пользователю.

Рис. 2.4. Визуальные элементы с идентификаторами

Определение ввода и вывода

Модуль *dash.dependencies* содержит в себе несколько классов, два из которых мы сейчас будем использовать. Это классы *Output* и *Input*.

Эти классы можно импортировать, добавив в начало кода строку, приведенную ниже:

```
from dash.dependencies import Output, Input
```

Давайте вспомним, что мы сделали на данный момент, перед тем как завершить задуманный функционал.

1. Создали экземпляр приложения в Jupyter Notebook.
2. Разместили в приложении выпадающий список с тремя элементами на выбор.
3. Написали функцию, возвращающую строковое значение выбранного элемента в списке.
4. Идентифицировали компоненты макета при помощи параметра `id`.
5. Импортировали классы `Input` и `Output` из модуля `dash.dependencies`.

Теперь пришло время определить функцию обратного вызова.

Функция обратного вызова (callback function) представляет собой декоратор, который в самом базовом виде требует указания следующих компонентов.

1. **Output**: элемент страницы, который будет изменен в результате запуска функции. Dash позволяет также указать, какое именно *свойство* (property) компонента подвергнется изменению. В нашем случае мы хотим изменить свойство `children` элемента `div`. Это можно выразить следующей строчкой кода:

```
Output(component_id='color_output', component_property='children')
```

2. **Input**: здесь используется та же логика – мы указываем элемент, который будет служить вводом для функции обратного вызова. В нашем случае это свойство `value` элемента `color_dropdown`:

```
Input(component_id='color_dropdown', component_property='value')
```

3. **Функция на языке Python**: здесь может быть использована любая функция, но, конечно, она должна выполнять что-то полезное в контексте элементов ввода и вывода.

На рис. 2.5 показан обновленный план выполнения нашего приложения.

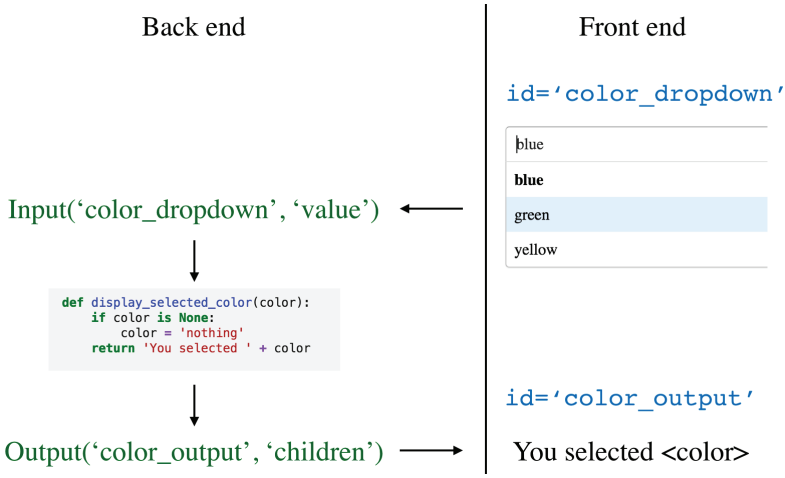


Рис. 2.5. Визуальные элементы приложения, связанные посредством свойств

> **Совет**
>
> Разграничение между клиентской и серверной частями в Dash для нас весьма условно. Обе они находятся в одном модуле, и нам нет необходимости заботиться о деталях. В данный момент мы можем рассматривать все, что располагается в `app.layout`, как клиентскую часть, а функции обратного вызова, определенные вне этого блока, – как серверную.

Шаблон функции обратного вызова

Общий формат создания функции обратного вызова состоит в ее определении в качестве атрибута `callback` переменной `app` с использованием точечной нотации классов Python и установке элементов ввода и вывода, как показано ниже:

```
@app.callback(Output(component_id, component_property), Input(component_id,
component_property))
```

Теперь, когда мы создали обратный вызов в виде атрибута объекта app и определили, какие компоненты и свойства должны влиять друг на друга, пришло время написать функцию на Python и поместить ее ниже:

```
@app.callback(Output(component_id, component_property), Input(component_id,
component_property))
def regular_function(input):
    output = do_something_with(input)
    return output
```

Вот мы и завершили создание шаблона функции обратного вызова и готовы перенести его в приложение.

Реализация функции обратного вызова

Давайте возьмем созданный ранее шаблон функции и наполним его смыслом:

```
@app.callback(Output('color_output', 'children'),
                Input('color_dropdown', 'value'))
def display_selected_color(color):
    if color is None:
        color = 'nothing'
    return 'You selected ' + color
```

Помните, что порядок в данном случае важен – сначала должно идти определение Output и только затем Input.

Теперь у нас есть полноценный обратный вызов. Ему прекрасно известно, какие свойства элементов вывода он будет модифицировать и какие элементы ввода использовать для этого. Для обработки данных используется функция display_selected_color, которая вычисляет результат и возвращает его компоненту с атрибутом ~~id~~, равным 'color_output', в котором он размещается в свойстве children.

Полный код приложения для запуска в JupyterLab и результаты нескольких попыток выбора цвета в выпадающем списке приведены на рис. 2.6.

Я также добавил в код элемент html.Br, преобразующийся в тег HTML
. Это сделано для лучшей читаемости.

Итак, мы создали свое первое небольшое, но вполне интерактивное приложение Dash. Мы проделали все шаги, досконально их разобрали и запустили приложение в JupyterLab. Конечно, это наше детище использует простейший набор данных и не может похвастаться богатым функционалом. И мы сделали это намеренно, чтобы вы могли полностью сосредоточиться на механизме добавления интерактивности. На самом деле вы понимаете, что нет никакой необходимости лишний раз повторять пользователю, какой именно цвет он только что выбрал в выпадающем списке.

Давайте с помощью приложения Dash постараемся ответить на более насущные вопросы, ответы на которые не лежат на поверхности.

```
from jupyter_dash import JupyterDash
import dash_core_components as dcc
import dash_html_components as html
from dash.dependencies import Output, Input

app = JupyterDash(__name__)

app.layout = html.Div([
    dcc.Dropdown(id='color_dropdown',
                 options=[{'label': color, 'value': color}
                          for color in ['blue', 'green', 'yellow']]),
    html.Br(),
    html.Div(id='color_output')
])

@app.callback(Output('color_output', 'children'),
              Input('color_dropdown', 'value'))
def display_selected_color(color):
    if color is None:
        color = 'nothing'
    return 'You selected ' + color

if __name__ == '__main__':
    app.run_server(mode='inline')
```

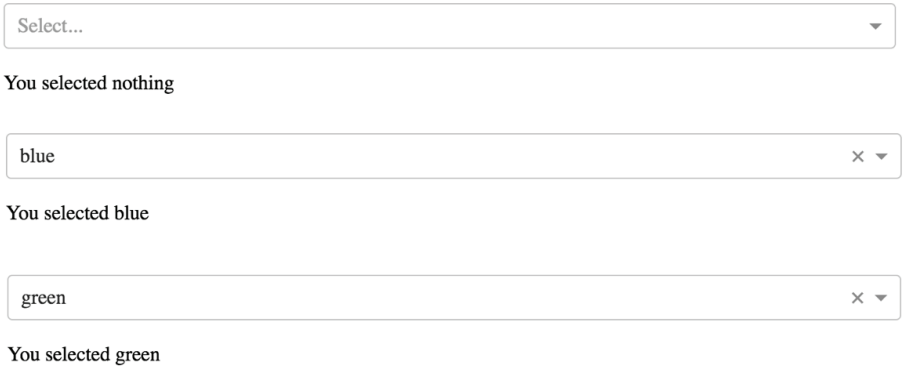


Рис. 2.6. Интерактивное приложение Dash в Jupyter Notebook

Внедрение функции в приложение

Давайте составим план функционала, который мы хотим воплотить в нашем приложении.

1. Создать выпадающий список со странами и регионами, имеющимися в нашем наборе данных.
2. Написать функцию обратного вызова, которая принимает на вход выбранную страну, фильтрует набор данных и ищет численность населения для этой страны за 2010 год.
3. Возвращает краткий отчет с результатами поиска. На рис. 2.7 показан желаемый вывод.

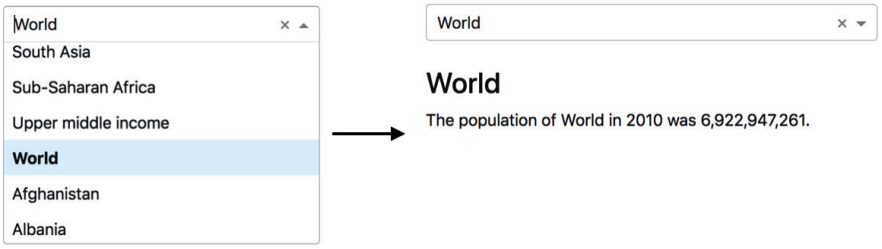


Рис. 2.7. Выпадающий список с поиском численности населения по стране

> **Примечание**
>
> Работая с наборами данных, мы будем открывать файлы, находящиеся в папке `data`. Предполагается, что ваше приложение располагается в той же директории, где и папка `data`. Исходные коды для каждой главы в хранилище GitHub помещены в соответствующие папки для облегчения доступа. При этом код будет исправно работать только в случае, если папка `data` и файл `app.py` располагаются в одной директории.

На рис. 2.8 показана предполагаемая иерархия файлов и папок.

```
dash_app/
├── app.py
└── data
```

Рис. 2.8. Структура файлов и папок приложения

Как мы и договаривались, будем тестировать минимальное приложение в JupyterLab, затем делать копию и добавлять новый функционал в основное приложение.

Для начала необходимо взглянуть на набор данных, немного его исследовать и понять, как можно реализовать новый функционал.

Для просмотра списка файлов в наборе данных можно запустить следующий код:

```
import os
os.listdir('data')

['PovStatsSeries.csv',
 'PovStatsCountry.csv',
 'PovStatsCountry-Series.csv',
 'PovStatsData.csv',
 'PovStatsFootNote.csv']
```

Если вам интересно, вы можете открыть файлы с данными и посмотреть их. В данный момент мы будем работать с файлом `PovStatsData.csv`. Чтобы понять его структуру, выполните следующий код:

```
import pandas as pd
poverty_data = pd.read_csv('data/PovStatsData.csv')
poverty_data.head(3)
```

Запустив этот код в JupyterLab, мы получим первые три строки из файла, показанные на рис. 2.9.

	Country Name	Country Code	Indicator Name	Indicator Code	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
0	East Asia & Pacific	EAS	Annualized growth in per capita real survey me...	SI.SPR.PC40.ZG	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	East Asia & Pacific	EAS	Annualized growth in per capita real survey me...	SI.SPR.PT10.ZG	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	East Asia & Pacific	EAS	Annualized growth in per capita real survey me...	SI.SPR.PT60.ZG	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Рис. 2.9. Первые несколько строк и столбцов набора данных

Похоже, у нас есть два столбца с фиксированными именами: **Country Name** и **Indicator Name**. Числовые показатели по годам (или отсутствующие значения NaN) располагаются в столбцах, имена которых совпадают с годами. У нас в распоряжении есть данные с 1974 по 2019 год, хотя на рис. 2.9 показан лишь их фрагмент. Страны и индикаторы также обладают кодами, которые могут нам пригодиться в дальнейшем, если мы захотим объединять различные датафреймы.

> **Совет**
>
> Фиксированные переменные – это переменные, известные нам заранее и неменяющиеся. В нашем случае это страны и индикаторы. Измеряемые переменные отражают значения, которые нас интересуют. Например, численность населения страны *A* в году *B*. Фиксированные переменные также называются измерениями. Чисто технически фиксированные и измеряемые переменные являются столбцами в наборе данных, и различия между ними исключительно концептуальные.

В главе 4 мы подробно поговорим о различных форматах данных и их влиянии на анализ и визуализацию. В данном случае было бы удобнее, если бы у нас была одна колонка с годами и одна со значениями, – это было бы привычнее для анализа. Но сейчас мы главным образом фокусируемся на функциях обратного вызова, так что будем работать с тем, что есть, чтобы сильно не отвлекаться.

Давайте реализуем наш план.

1. Создадим выпадающий список. Для этого можно воспользоваться методом `Series.unique` из пакета `pandas` для удаления дубликатов по странам и регионам. Ниже создадим пустой элемент `div` со свойством `id='report'`:

```
dcc.Dropdown(id='country',
                options=[{'label': country, 'value': country}
                            for country in
                            poverty_data['Country Name'].unique()])
html.Div(id='report')
```

2. Напишем функцию обратного вызова, которая будет принимать выбранную страну, фильтровать набор данных и выполнять поиск численности населения по этому городу за 2010 год. Фильтрация будет выполняться в два этапа.

Сначала нужно проверить, не остался ли пустым выбор страны в списке. Такое может случиться, если пользователь только что зашел на страницу и еще не успел сделать выбор или отменил ранее сделанный выбор в списке. В этом случае мы просто возвращаем пустую строку:

```
if country is None:
return ''
```

Теперь перейдем к самой фильтрации. Возьмем выбранную в выпадающем списке страну и отфильтруем по ней датафрейм `poverty_data`, получив численность ее населения по годам. В результате получим датафрейм, в котором останутся строки с выбранной страной в столбце **Country Name** и значением **Population, total** в столбце **Indicator Name**. Назовем его `filtered_df`.

После этого воспользуемся методом `loc` для получения всех строк из колонки с именем **2010** и извлечем первый элемент при помощи атрибута `values`, как показано ниже:

```
filtered_df = poverty_data[(poverty_data['Country Name']==country) &
                    (poverty_data['Indicator Name']=='Population, total')]
population = filtered_df.loc[:, '2010'].values[0]
```

Наконец, сформируем краткий отчет о выполненной работе. Получив нужное значение, мы вернем список из двух элементов. В первом будет находиться элемент `<h3>` с демонстрацией содержимого переменной `country` в виде заголовка третьего уровня, а во втором – предложение в виде *f-строки* с двумя динамическими переменными, что видно ниже:

```
return [
        html.H3(country),
        f'The population of {country} in 2010 was {population:,.0f}.'
]
```

Заметьте, что поскольку у нас в макете уже размещен элемент `div` и мы объявили в функции обратного вызова, что она будет изменять его свойство `children` (которое может принимать отдельное значение или список), возвращаемое значение функции может быть как списком, так и скаляром.

Также мы отформатировали значение переменной `population` в возвращаемом списке для облегчения чтения. Здесь двоеточие говорит о том, что далее последует строка форматирования. Запятая означает, что именно с помощью нее будут разделяться тысячи, точка отвечает за форматирование десятичных разрядов, а ноль после нее указывает количество десятичных разрядов. Символ `f` на конце говорит о том, что мы имеем дело с типом данных `float`.

Теперь мы готовы провести рефакторинг кода, включив в него визуальные элементы и новый функционал.

Если помните, в коде нашего приложения был такой фрагмент:

```
html.H2('The World Bank'),
dbc.Tabs([
```

Давайте между этими двумя строками поместим новый фрагмент, приведенный ниже:

```
dcc.Dropdown(id='country',
                options=[{'label': country, 'value': country}
                    for country in poverty_data['Country Name'].unique()]),
html.Br(),
html.Div(id='report'),
html.Br(),
```

Функцию обратного вызова мы разместим после закрывающей скобки корневого элемента `html.Div`. Ниже приведен код функции:

```
@app.callback(Output('report', 'children'),
                Input('country', 'value'))
def display_country_report(country):
  if country is None:
      return ''

  filtered_df = poverty_data[(poverty_data['Country Name']==country) &
                             (poverty_data['Indicator Name']=='Population, total')]
  population = filtered_df.loc[:, '2010'].values[0]

  return [html.H3(country),
      f'The population of {country} in 2010 was {population:,.0f}.']
```

Запустив приложение, вы увидите его обновленный вид, показанный на рис. 2.10.

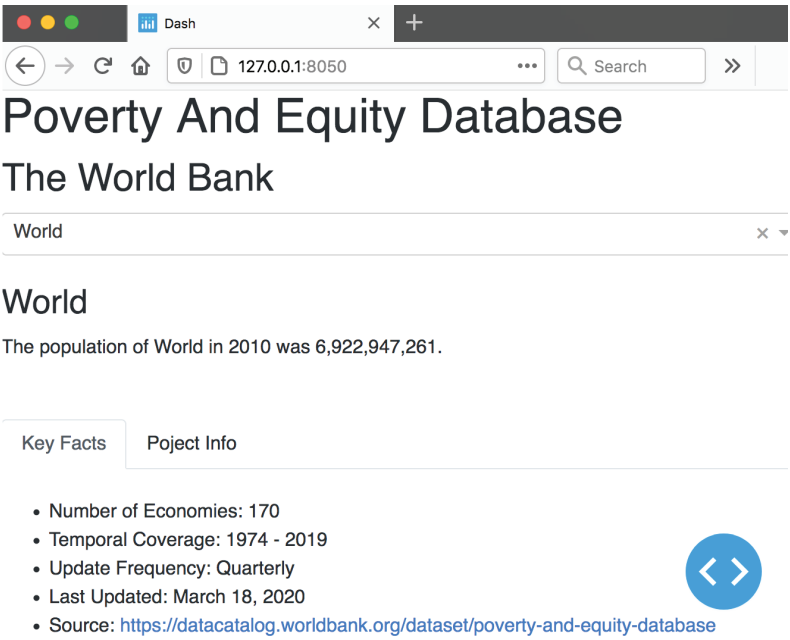


Рис. 2.10. Обновленное приложение с выпадающим списком и выводом численности населения

> **Совет**
>
> Метод `app.run_server` принимает необязательный параметр `port`, который по умолчанию равен **8050**, что видно в адресной строке браузера на рис. 2.10. В процессе разработки вам иногда может понадобиться запускать одновременно два и более приложений. Это можно сделать, запуская новые приложения на других портах. Например, вы можете написать `app.run_server(port=1234)`. Это применимо и к объекту `jupyter_dash`.

Полный код приложения:

```
import dash
import dash_html_components as html
import dash_core_components as dcc
import dash_bootstrap_components as dbc
from dash.dependencies import Output, Input
import pandas as pd

app = dash.Dash(__name__, external_stylesheets=[dbc.themes.BOOTSTRAP])

poverty_data = pd.read_csv('data/PovStatsData.csv')

app.layout = html.Div([
    html.H1('Poverty And Equity Database'),
```

```
    html.H2('The World Bank'),
    dcc.Dropdown(id='country',
                 options=[{'label': country, 'value': country}
                          for country in poverty_data['Country Name'].unique()]),
    html.Br(),
    html.Div(id='report'),
    html.Br(),
    dbc.Tabs([
       dbc.Tab([
           html.Ul([
               html.Br(),
               html.Li('Number of Economies: 170'),
               html.Li('Temporal Coverage: 1974 - 2019'),
               html.Li('Update Frequency: Quarterly'),
               html.Li('Last Updated: March 18, 2020'),
               html.Li([
                   'Source: ',
                   html.A('https://datacatalog.worldbank.org/dataset/poverty-
and-equity-database',
                          href='https://datacatalog.worldbank.org/dataset/
poverty-and-equity-database')
               ])
           ])

       ], label='Key Facts'),
        dbc.Tab([
            html.Ul([
                html.Br(),
                html.Li('Book title: Interactive Dashboards and Data Apps
with Plotly and Dash'),
                html.Li(['GitHub repo: ',
                         html.A('https://github.com/PacktPublishing/
Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash',
                                href='https://github.com/PacktPublishing/
Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash')
                         ])
            ])
        ], label='Project Info')
    ]),

])


@app.callback(Output('report', 'children'),
              Input('country', 'value'))
def display_country_report(country):
    if country is None:
        return ''

    filtered_df = poverty_data[(poverty_data['Country Name']==country) &
                               (poverty_data['Indicator Name']=='Population, total')]
```

```
    population = filtered_df.loc[:, '2010'].values[0]

    return [html.H3(country),
            f'The population of {country} in 2010 was {population:,.0f}.']


if __name__ == '__main__':
    app.run_server(debug=True)
```

Теперь, когда мы создали функцию обратного вызова и внедрили ее в приложение, можно посмотреть, как использовать загадочную синюю *кнопку отладки*, располагающуюся при запуске приложения справа внизу.

Если нажать на эту кнопку и выбрать режим **Callbacks**, мы увидим интерактивную диаграмму, отображающую компоненты в том порядке, в котором они определены: **country** и его **value**, а затем **report** и его **children**, как показано на рис. 2.11.

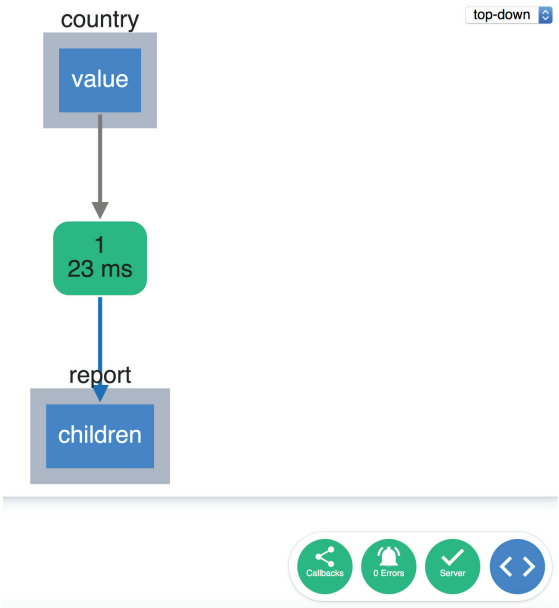


Рис. 2.11. Визуальная отладка приложения Dash в действии

Кнопка **Server** горит зеленым цветом, что означает нормальную работу приложения. Также мы видим сообщение *0 Errors* на соседней кнопке, перевод которого не требуется. Если это окно отладки остается открытым в процессе работы приложения, вы увидите, что взаимодействие с компонентами будет приводить к обновлению путей обратных вызовов. Компоненты, участвующие в вызове, подсвечиваются, чтобы вы понимали, что происходит с приложением в данный момент. Это бывает очень удобно при отладке действительно сложных приложений. Узлы на диаграмме также являются интерактивными – вы можете смело перемещать их по окну, уменьшая/увеличивая размеры диаграммы, чтобы вам было удобнее просматривать активные связи. И да, окно

отладки также является приложением Dash, в котором используются пакеты этого фреймворка.

Теперь обратите внимание на зеленый объект в центре диаграммы – на нем расположены две важные цифры. Первая (вверху), показывающая единицу, характеризует количество раз использования этой связи. Цифра внизу символизирует время, которое потребовалось на выполнение обратного вызова. Эти показатели бывают весьма полезны при анализе работы приложения.

До сих пор мы использовали элементы ввода с единственным значением (а не списки, например). А что, если нам необходимо некоторым образом обработать сразу несколько значений? Или мы хотим передать на вход функции значения из разных элементов – допустим, из выпадающего списка и календаря. Все это возможно реализовать в Dash с помощью тех же функций обратного вызова. Я уже упоминал, что они являются сердцем этого фреймворка?

Думаю, мы уже написали достаточно много кода в этом разделе. Пришло время узнать о дополнительных возможностях, предлагаемых функциями обратного вызова, и полезных свойствах, о которых мы еще не упоминали. Пока мы просто перечислим все эти важные и интересные особенности, а поработаем с ними в следующих главах книги.

Свойства функций обратного вызова

Давайте пройдемся по свойствам и особенностям функций обратного вызова Dash, со многими из которых будем работать в следующих главах:

- **множественные элементы ввода**: как мы уже упоминали, можно передавать на вход функции обратного вызова более одного элемента ввода, тем самым реализуя более сложный функционал приложения. К примеру, в нашем приложении мы могли бы предоставить пользователю возможность выбрать не только страну, но также год и экономический показатель для анализа. Все три элемента ввода могли быть использованы в процессе фильтрации исходного датафрейма, что позволило бы вернуть значение на основе сразу нескольких критериев;
- **элементы ввода могут быть списками** (возможность передать несколько значений посредством одного элемента ввода): к примеру, выпадающий список со странами можно настроить на множественный выбор, что позволит пройти по всем выбранным странам и визуализировать информацию по ним как на одной диаграмме, так и на разных;
- **множественные элементы вывода**: как и в случае с элементами ввода, функция обратного вызова может быть связана сразу с несколькими элементами вывода. Допустим, мы могли бы отправить отфильтрованные данные как на диаграмму, так и в таблицу – на случай, если пользователю понадобятся сырые данные для экспорта или дальнейшего анализа;
- **функции обратного вызова могут делать что угодно**: мы сказали о том, что функции обратного вызова получают данные из элементов ввода, обрабатывают их и отправляют на элементы вывода. Но не забывайте, что это прежде всего функции, а значит, они могут выполнять

множество разных действий, например отправку электронной почты. Также полезной возможностью функций является журналирование. Все, что вам нужно сделать, – это записывать логи по аргументам, переданным функции. Это позволит понять, что интересно пользователям, какой функционал они используют и т. д. Вы можете даже анализировать эти логи в отдельном приложении, если необходимо;

- **порядок имеет значение**: элементы вывода должны следовать перед элементами ввода. Кроме того, порядок следования элементов ввода должен строго соответствовать порядку передаваемых в функцию аргументов. Рассмотрим следующий пример:

```
@app.callback(Output('div_1', 'children'),
                Input('dropdown', 'value'),
                Input('date', 'value)
def my_function(dropdown, date):
    output = process(dropdown, date)
    return output
```

Здесь первый элемент `Input` в декораторе функции должен соответствовать первому параметру функции `my_function`. В этом примере мы использовали одинаковые имена соответствующих параметров (`dropdown` и `date`) для большей наглядности. То же касается и элементов вывода;

- **State**: еще один необязательный параметр функций обратного вызова. В примерах, которые мы рассматривали до сих пор, обратные вызовы срабатывали непосредственно после изменения значений. Но иногда это не требуется. Допустим, если у функции несколько элементов ввода, пользователю может не понравиться, что вывод постоянно обновляется по мере произведения настроек. Представьте, что в приложении есть текстовое поле, изменение которого влияет на другие элементы. И зачем нам нужно, чтобы после ввода каждого очередного символа происходило обновление информации на странице? Обычно параметр `State` используется с кнопками. Пользователь выбирает или вводит значение и по готовности нажимает на кнопку, которая и инициирует обратный вызов.

На рис. 2.12 показана концептуальная диаграмма более сложной функции обратного вызова.

Итак, мы создали и запустили две функции обратного вызова в двух разных контекстах. Одну из них мы также внедрили в рабочий проект из главы 1. Если вы еще немного потренируетесь, то сможете без проблем писать собственные функции обратного вызова. Вместе с тем продолжайте учиться производить рефакторинг кода, развивая приложения без нарушения ранее созданного функционала.

Давайте вспомним все, чему мы научились в этой главе.

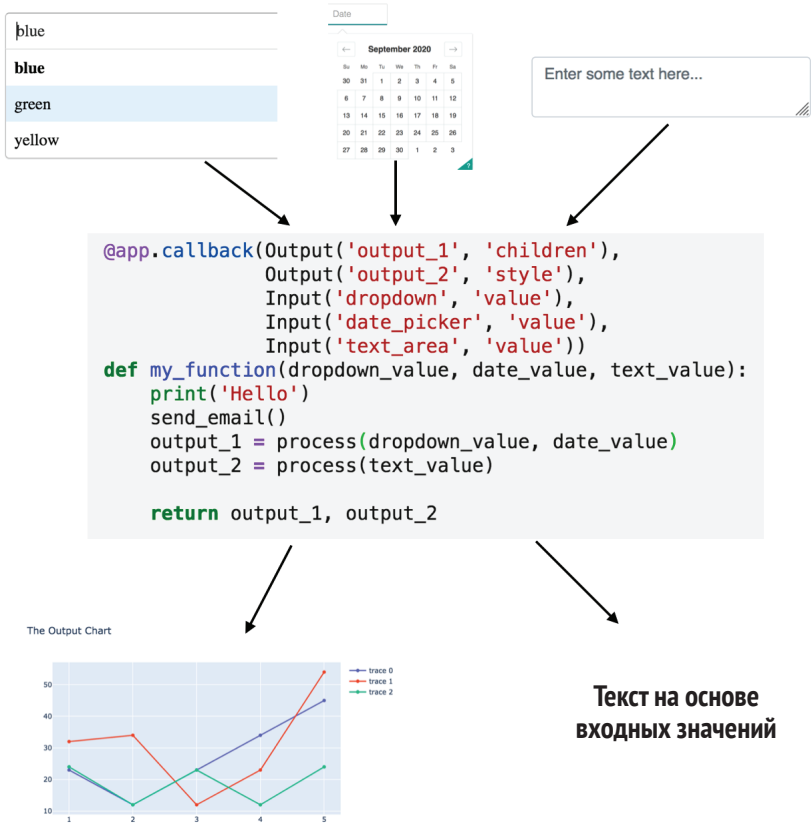


Рис. 2.12. Функция обратного вызова со множеством элементов ввода и вывода

Заключение

Первое, что мы сделали, – это научились запускать приложения Dash в среде Jupyter Notebook. Вы увидели, что этот процесс не включает в себя ничего нового, все знакомо и понятно. Мы даже вместе создали первое интерактивное приложение в ноутбуке. Мы детально проговорили все этапы разработки приложения: от создания компонентов макета, присвоения им идентификаторов, выбора свойств, которые будут использоваться, и до связывания их с функциями обратного вызова. После этого мы создали еще одно приложение на основе полноценного набора данных. Мы научились внедрять новый функционал в приложение и создали простой интерактивный отчет о численности населения стран. Мои поздравления!

В следующей главе мы начнем путешествие по миру визуализации Plotly. Мы познакомимся с объектом **Figure** и его компонентами, а также узнаем, как к ним можно обращаться и как модифицировать. Это позволит нам получить полный контроль над создаваемыми визуализациями.

Глава 3

Работа с объектом Figure

Представьте, что вы опубликовали статью с графиком. И допустим, что читатели в среднем будут рассматривать ваш график около одной минуты. Если вы сделали свою диаграмму простой и доступной, они секунд 10 потратят на ее понимание, а оставшиеся 50 секунд будут думать, анализировать и делать выводы. Если же ваш график окажется слишком сложным для понимания, читатели будут тратить 50 секунд на его чтение, а на анализ и выводы времени у них просто не останется.

Эта глава будет посвящена инструментам, которые вы сможете использовать с целью минимизации времени, требуемого для понимания ваших графиков. Если в двух предыдущих главах мы в основном уделяли внимание структуре приложения и его интерактивности, то сейчас акцентируем свой взгляд на управлении самими визуальными объектами, вокруг которых строится приложение. И главным образом мы будем говорить об объекте **Figure**, входящем в состав Plotly. Темы, которые будут рассмотрены:

- введение в объект `Figure`;
- знакомство с атрибутом `data`;
- знакомство с атрибутом `layout`;
- ряды данных и способы их добавления на график;
- способы преобразования графиков.

Технические требования

В этой главе мы начнем работать с объектом `Figure` из модуля `graph_objects` пакета `plotly`. Позже мы добавим остальные пакеты, с которыми вы уже знакомы по первым главам, а именно Dash, Dash HTML Components, Dash Core Components, Dash Bootstrap Components, JupyterLab, Jupyter Dash и pandas.

Все эти пакеты могут быть установлены отдельно при помощи инструкции `pip install <package-name>`, но для чистоты эксперимента будет неплохо, если вы установите те версии пакетов, которые использовались при написании книги. Это можно сделать, запустив из корневой папки репозитория команду `pip install -r requirements.txt`. Актуальный набор данных `poverty` можно загрузить по адресу **https://datacatalog.worldbank.org/dataset/poverty-and-equity-database**. Однако, как и в случае с пакетами, вы можете использовать оригинальный набор данных, находящийся в папке `data` в корне репозитория Git. Весь код примеров

из этой главы вы сможете найти на GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_03**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3x9VhAA**.

Введение в объект Figure

Plotly представляет собой полноценную систему визуализации данных с более чем 50 встроенными типами диаграмм (столбчатые, точечные, гистограммы и т. д.). Она поддерживает двумерные и трехмерные диаграммы, *тернарные графики* (ternary plot), географические карты и многое другое. При этом свобода действий при настройке графических объектов практически ничем не ограничена, и можно бесконечно долго выбирать наиболее подходящий своим требованиям вид визуализации. Это те муки выбора, с которыми не против был бы столкнуться любой разработчик.

Обычно мы используем визуальные элементы тогда, когда хотим продемонстрировать какие-то закономерности или связи между имеющимися данными. Но визуализация будет абсолютно бесполезна, если вы не знаете, что хотите сказать. Представьте прямоугольник с хаотично разбросанными по нему точками. Если вы не знаете, что находится на осях, для вас эта картинка будет лишена всякого смысла. Если точки заменить на маркеры определенной формы, вы все равно ничего не поймете без соответствующей легенды. Также нам обычно нужны заголовки и подписи для осознания общего контекста представленной информации.

Двумя верхнеуровневыми атрибутами объекта `Figure` являются `data` и `layout`, последний из которых представляет собой коллекцию вспомогательных элементов. Каждый атрибут обладает целой иерархией вложенных атрибутов, что образует древовидную структуру. Также стоит отметить атрибут `frames`, который главным образом используется для анимации и не так широко распространен, как первые два, представленные на каждом графике. Мы не будем обсуждать его в этой главе.

Давайте подробно рассмотрим первые два атрибута и уже построим какой-нибудь график, чтобы лучше понимать, как они связаны с объектом `Figure`:

- `data`: различные атрибуты данных и отношения между ними выражаются при помощи графических/геометрических фигур, например окружностей, прямоугольников, линий и пр. Графические атрибуты этих фигур используются для выражения различных атрибутов данных. Относительный размер, длина и расстояние между фигурами – это то, что мы можем видеть. Визуальная, а значит, интуитивная природа этих атрибутов делает их хорошо понятными, и дополнительных объяснений по ним не требуется. Атрибут `data` соответствует сути той информации, которую вы пытаетесь извлечь из графика. Значения, передаваемые этому атрибуту, напрямую зависят от типа используемой диаграммы. Например, для диаграммы рассеяния вы должны передать значения `x` и `y`. Для географической карты – `lat` и `lon`. Кроме того, вы можете наслаивать друг на друга данные в рамках одной диаграммы. В таком случае у вас на

графике будет присутствовать несколько *рядов данных* (trace). Каждый отдельный тип диаграммы предусматривает передачу дополнительных необязательных параметров, о многих из которых мы расскажем в этой и следующих главах книги;

- `layout`: к этому атрибуту относится все, что не касается непосредственно данных. Элементы этого атрибута более абстрактны по своей природе. Обычно в качестве выразительных средств здесь используется сопроводительный текст, дающий пользователю понять, с чем он имеет дело. Также сюда относятся элементы стилизации, не говорящие о данных напрямую, но помогающие лучше понять смысл диаграммы и быстрее прийти к верному выводу. Мы будем говорить о многих атрибутах этой группы, включая заголовки графиков и осей, шкалы и легенды. Эти элементы включают в себя вложенные атрибуты, такие как размер шрифта, расположение и т. д.

Как известно, лучше всего учиться на примерах, и мы прямо сейчас создадим свой первый объект `Figure`. Модуль `graph_objects` обычно импортируется с псевдонимом `go`, так что создание экземпляра объекта `Figure` будет выглядеть так: `go.Figure`. На рис. 3.1 показан процесс создания и внешний вид пустого объекта `Figure`.

```
1  import plotly.graph_objects as go
2
3  go.Figure()
```

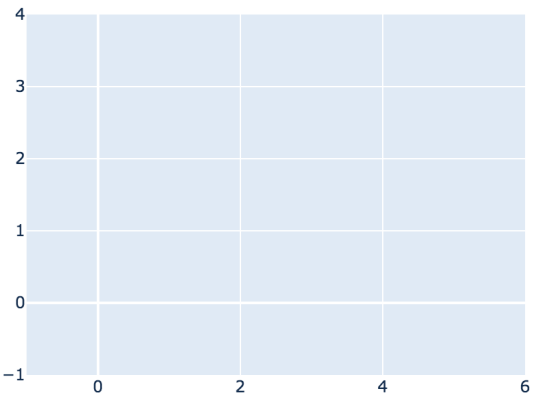


Рис. 3.1. Пустой объект Figure, отображенный в JupyterLab

Конечно, по пустому графику мы мало что сможем понять, но это был первый шаг, и теперь мы готовы наполнять нашу визуализацию смыслом. Мы могли бы определить все характеристики объекта `Figure` непосредственно при его создании, но мы пойдем более простым и понятным путем – сохраним созданный объект в переменную, после чего последовательно добавим и/или изменим его характеристики. Важным преимуществом такого подхода является

возможность менять атрибуты и поведение графического объекта после его создания.

> **Важно**
>
> После присвоения объекта `Figure` переменной она будет принадлежать глобальной области видимости. А поскольку она является изменяемой, вы можете модифицировать ее в других участках кода. Отображение объекта после выполнения этих изменений сделает видимым все внесенные правки. Мы воспользуемся этой особенностью для управления нашими графическими объектами.

Итак, мы создали базовый графический объект и теперь готовы разместить на нем наш первый ряд данных.

Знакомство с атрибутом data

Давайте начнем с простой *диаграммы рассеяния* или *точечной диаграммы* (scatter plot) на основе небольшого и очень незамысловатого набора данных. Далее в этой главе мы продолжим работать с этим датасетом. После создания объекта `Figure` и присвоения его переменной в вашем распоряжении оказывается великое множество инструментов для дальнейшей работы с ним. Названия всех методов, служащих для добавления рядов данных на диаграмму, начинаются с `add_`, после чего следует тип диаграммы. Например, `add_scatter` или `add_bar`.

Пройдем вместе весь процесс создания диаграммы рассеяния.

1. Импортируйте модуль `graph_objects`, как показано ниже:

   ```
   import plotly.graph_objects as go
   ```

2. Создайте объект `Figure` и присвойте его переменной:

   ```
   fig = go.Figure()
   ```

3. Добавьте ряд данных. Для этого типа диаграммы минимально допустимым является набор из двух параметров `x` и `y` в виде массивов. Также эти параметры могут быть переданы в виде *списков* (list), *кортежей* (tuple), массивов NumPy или объектов Series:

   ```
   fig.add_scatter(x=[1, 2, 3], y=[4, 2, 3])
   ```

   Выведите созданный график. Вы можете просто написать имя переменной в последней ячейке с кодом в JupyterLab, и график будет отображен при запуске. Также вы можете явным образом вызвать метод `show`, что даст вам дополнительные возможности для настройки отображения графика:

   ```
   fig.show()
   ```

Полный код приложения и вывод диаграммы приведен на рис. 3.2.

```
1  import plotly.graph_objects as go
2
3  fig = go.Figure()
4  fig.add_scatter(x=[1, 2, 3], y=[4, 2, 3])
5  fig.show()
```

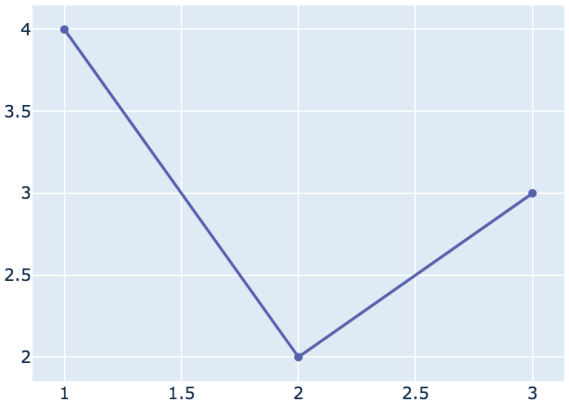


Рис. 3.2. Точечная диаграмма в JupyterLab

Теперь добавим еще один ряд данных поверх существующего графика. Для этого просто повторим третий шаг, но с другими значениями:

```
fig.add_scatter(x=[1, 2, 3, 4], y=[4, 5, 2, 3])
```

Это приведет к добавлению еще одной линии на графике. Если инструкцию `fig.show()` запустить после этой строки, то вы увидите обновленную диаграмму. Заметьте, что в этом ряду данных четыре точки, тогда как в первом было три. Но об этом не стоит беспокоиться, эта ситуация будет решена при помощи значений по умолчанию. Мы также можем изменить эти значения по умолчанию, если потребуется.

Если нам нужно изменить какие-то аспекты отображения диаграммы, мы можем сделать это при вызове метода `add_<chart_type>`. Эти методы позволяют передать большое количество параметров, зависящих от конкретного типа диаграммы. Во второй части книги мы подробнее поговорим о некоторых из этих методов и рассмотрим параметры, которые они принимают. Что касается атрибута `layout`, мы можем менять его вложенные элементы при помощи привычной для Python точечной нотации, например так: `figure.attribute.sub_attribute = value`. Это не совсем правильная инструкция, поскольку существует несколько исключений, когда возникают пересечения с тем, что атрибут, к примеру, принадлежит атрибуту `data`, а управляется из `layout`. Об этом стоит помнить.

Давайте посмотрим, что мы можем менять в атрибуте `layout`.

Знакомство с атрибутом layout

Давайте для нашего объекта `Figure` добавим общий заголовок, а также заголовки осей, как показано ниже:

```
fig.layout.title = 'The Figure Title'
fig.layout.xaxis.title = 'The X-axis title'
fig.layout.yaxis.title = 'The Y-axis title'
```

Как видите, объект `Figure` обладает древовидной структурой. К примеру, атрибут `title` присутствует как у объекта `fig.layout`, так и у `fig.layout.xaxis` с `fig.layout.yaxis`. Чтобы вы понимали, насколько разветвленное это дерево, на рис. 3.3 показаны лишь некоторые атрибуты объекта `xaxis`, начинающиеся с `tick`.

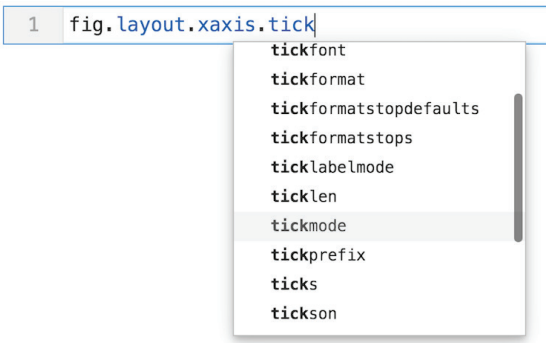


Рис. 3.3. Некоторые атрибуты объекта `layout.xaxis`

Теперь давайте взглянем на результат добавления последних строк кода в наше приложение.

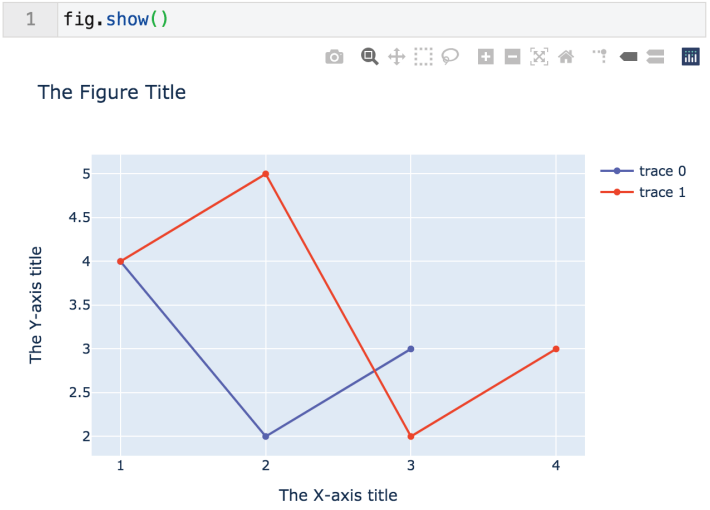


Рис. 3.4. Обновленный график с двумя рядами данных, легендой и заголовками

Добавленные заголовки, показанные на рис. 3.4, не нуждаются в дополнительном пояснении. Линии получили цвета по умолчанию, хорошо отличимые друг от друга. Отдельно стоит коснуться легенды, которая была добавлена автоматически. При наличии одного ряда данных в легенде, разумеется, нет никакой необходимости, но если линий на диаграмме несколько, без нее бывает не обойтись. В этом смысле очень важно давать рядам данных осмысленные имена. Конечно, имена `trace 0` и `trace 1` мало о чем говорят, но мы оставим их как есть, чтобы отличать наши линии.

График, который мы с вами построили, увидят пользователи приложения. Давайте взглянем на компоненты нашего объекта `Figure`.

Интерактивное исследование объекта Figure

Как мы уже упоминали ранее, метод `show` предлагает удобные возможности для настройки отображения объекта `Figure`. В частности, вы можете установить значение параметра `renderer` равным `'json'`. На рис. 3.5 показано, к чему это приведет.

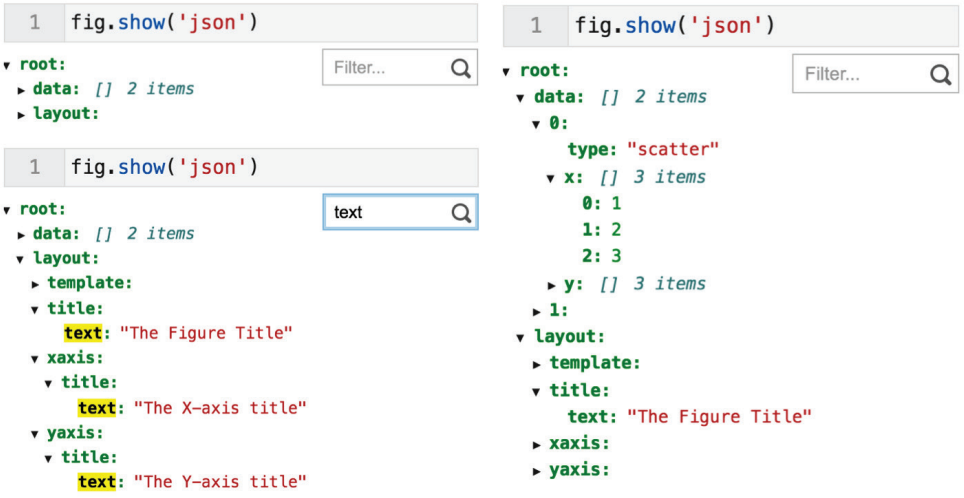


Рис. 3.5. Интерактивное исследование объекта Figure в JupyterLab

В левом верхнем углу рис. 3.5 вы видите отображение по умолчанию. Элемент **root** представляет объект `Figure`, а под ним располагаются два его главных атрибута. Также в правой части рисунка видно, что атрибут **data** содержит в себе два элемента, соответствующих нашим рядам данных. Треугольники слева от элементов указывают на то, открыты они в данный момент или нет.

В левом нижнем углу рисунка показано, как работает функция поиска по объекту. Это бывает очень полезно, когда вам необходимо получить доступ или изменить какой-то конкретный атрибут, но вы не уверены, как именно он называется или какому элементу принадлежит. Я раскрыл пару элементов, чтобы вы видели, что значения атрибутов соответствуют заголовкам на нашем графике.

> **Важно**
>
> Эта глава, да и вся книга в целом, посвящена созданию диаграмм и дашбордов. В ней мы не будем обсуждать принципы визуализации данных или статистического анализа. Иными словами, мы сосредоточимся на том, чтобы вы могли создать тот график, который хотите, а не тот, который следовало бы. По ходу дела мы будем знакомить вас с правилами хорошего тона в вопросах визуализации, но важно помнить, что это не главная цель для книги.

Уверен, вы заметили в верхней части диаграммы *панель инструментов* (mode bar), содержащую ряд кнопок. У метода `show` есть полезный параметр `config`, который, в частности, отвечает за то, какие кнопки показывать на панели, а какие скрывать.

Опции настройки для объекта Figure

Параметр `config` метода `show` принимает на вход словарь, с помощью которого осуществляются некоторые настройки объекта `Figure`. Ключи в словаре отвечают за аспекты настройки. Значения могут представлять собой строки или списки – в зависимости от того, что именно вы настраиваете. Рассмотрим следующий пример:

```
fig.show(config={'displaylogo': False,
                 'modeBarButtonsToAdd': ['drawrect',
                                         'drawcircle',
                                         'eraseshape']})
```

Ниже перечислены наиболее важные аспекты настройки:

- `displayModeBar`: по умолчанию этот атрибут имеет значение `True`. Он отвечает за то, показывать или нет панель инструментов;
- `responsive`: этот атрибут также по умолчанию равен `True`, и с помощью него можно указать, должны ли меняться размеры графика в зависимости от размера окна браузера. Иногда бывает полезно зафиксировать внешний вид диаграммы;
- `toImageButtonOptions`: иконка с изображением камеры на панели инструментов позволяет пользователю загрузить графический объект как картинку. С помощью этого атрибута контролируются опции загрузки по умолчанию. Значением для атрибута является словарь, в котором вы можете задать формат по умолчанию (SVG, PNG, JPG или WebP). Вы также можете задать значения по умолчанию для высоты, ширины, масштаба и имени файла;
- `modeBarButtonsToRemove`: в этом атрибуте содержится список кнопок, которые вы не хотите видеть на панели инструментов.

Теперь, когда вы знаете, как создавать и настраивать основные виды диаграмм, давайте посмотрим, что еще с ними можно сделать. К примеру, как

преобразовать их в другие форматы? И в какие вообще форматы можно их преобразовывать?

Способы преобразования графиков

Названия методов, отвечающих за конвертацию объекта `Figure` в другие форматы, начинаются с `to_` или `write_`. Давайте рассмотрим самые популярные форматы.

Преобразование графиков в HTML

Графики Plotly по своей сути представляют объекты HTML, за интерактив в которых отвечает JavaScript. Таким образом, мы легко можем преобразовать их в файлы HTML, чтобы поделиться ими, например, по электронной почте. Вы должны подразумевать такую возможность при создании дашбордов. Пользователь может построить график или отчет, после чего преобразовать его в HTML-формат, скачать и поделиться с коллегами.

Все, что вам нужно сделать, – это передать путь, по которому будет сохранен файл, в метод `write_html`. Также вы можете передать дополнительные необязательные параметры для тонкой настройки функционала. Давайте преобразуем наш график в HTML и с помощью параметра `config` укажем формат файла SVG. Результат будет виден при нажатии на иконку с камерой. Код получился очень простым:

```
fig.write_html('html_plot.html',
               config={'toImageButtonOptions':
                       {'format': 'svg'}})
```

Теперь можно открыть сохраненный файл HTML на всю ширину браузера, как показано на рис. 3.6.

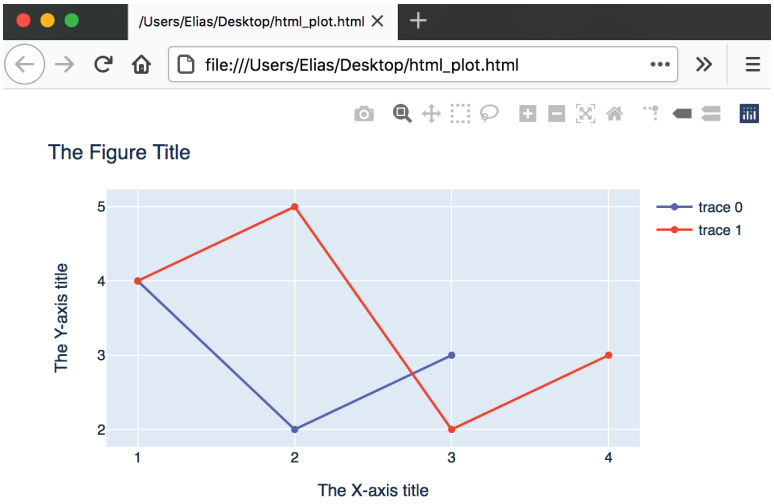


Рис. 3.6. Объект Figure в виде отдельного файла HTML в браузере

Преобразование графиков в изображения

Мы рассмотрели способ, позволяющий пользователю вручную загрузить изображение объекта `Figure`. Но есть и другой подход, с помощью которого можно сделать это программно. Подобно методу `write_html`, вы можете воспользоваться методом `write_image`. При этом формат изображения может быть указан явно или определен исходя из расширения переданного файла. Также вы можете задать значения параметров `height` и `width`.

Это может быть удобно при массовом создании изображений. Например, у вас может быть множество графиков – каждый для своей страны, и вам может понадобиться загрузить их все отдельно для разных отчетов. Вручную сделать это будет проблематично. Вы могли бы также задействовать одну из функций обратного вызова. После построения соответствующих отчетов пользователь, к примеру, может нажать на кнопку для преобразования графиков в изображения и их загрузки. Это можно реализовать в виде HTML-конвертера:

```
fig.write_image('path/to/image_file.svg',
                    height=600, width=850)
```

Теперь пришло время перейти к практике и внимательнее присмотреться к нашему набору данных.

Работа с настоящим набором данных

В главе 2 мы создали простой отчет о численности населения выбранной страны в 2010 году. Это нормально, если пользователь точно знает, что хочет увидеть. То есть у него есть четкое понимание о том, какой именно показатель и за какой точно период по странам он хочет посчитать. Наш отчет предоставляет ему такую возможность.

Вообще, дашборды можно условно разделить на две категории. В первую входят дашборды, отвечающие на конкретный вопрос, а во вторую – обладающие определенным исследовательским потенциалом. В этом случае пользователь недостаточно много знает про исследуемую область, и ему нужен общий обзор.

При этом в процессе работы пользователи могут переключаться между этими двумя категориями дашбордов. К примеру, сначала они могут исследовать показатели бедности за минувшее десятилетие, выявить регион с выделяющимися на фоне остальных показателями, после чего задать конкретный вопрос относительно этого региона. Поняв, что в данном регионе выделяется не только этот показатель, они могут перейти к новому исследованию с целью получить больше информации.

Итак, мы хотим дать пользователю возможность выбрать год и увидеть первые 20 стран по численности населения за этот год.

Как вы помните, в нашем датасете присутствует информация о стране, включая ее название и код, о показателе (также с кодом), а также колонки со значениями по годам – с 1974-го по 2019-й.

Как договаривались, давайте сначала решим задачу в изолированном окружении в JupyterLab.

1. Импортируйте пакет `pandas`, используйте его для открытия нашего датасета и присвойте его переменной `poverty_data`, как показано ниже:

```
import pandas as pd
poverty_data = pd.read_csv('data/PovStats_csv/PovStatsData.csv')
```

2. Хотя столбец, который нас интересует, называется **Country Name**, в нем также содержится информация о регионах, что может быть полезно, но не в нашем случае. Мы можем выделить регионы в список, чтобы впоследствии отфильтровать по нему набор данных. Я вручную скопировал перечень регионов, создав тем самым следующий список:

```
regions = ['East Asia & Pacific', 'Europe & Central Asia', 'Fragile
and conflict affected situations', 'High income', 'IDA countries
classified as fragile situations', 'IDA total', 'Latin America &
Caribbean', 'Low & middle income', 'Low income', 'Lower middle income',
'Middle East & North Africa', 'Middle income', 'South Asia',
'Sub-Saharan Africa', 'Upper middle income', 'World']
```

3. Создайте переменную `population_df`, оставив в ней строки исходного датафрейма, в которых значения поля **Country Name** не входят в список `regions`, а в колонке **Indicator Name** содержится значение **Population, total**. Метод `isin` принадлежит объекту `Series` и позволяет проверить, входят ли значения серии в заданный список. При помощи логического оператора ~ (тильда) формулируется отрицание:

```
population_df = poverty_data[~poverty_data['Country Name'].
isin(regions) & (poverty_data['Indicator Name']== 'Population, total')]
```

4. Первые несколько строк можно вывести следующим образом:

```
population_df.head()
```

Картина должна быть такой, как на рис. 3.7.

	Country Name	Country Code	Indicator Name	Indicator Code	1974	1975	1976	1977	1978
1051	Afghanistan	AFG	Population, total	SP.POP.TOTL	12412950.0	12689160.0	12943093.0	13171306.0	13341198.0
1115	Albania	ALB	Population, total	SP.POP.TOTL	2350124.0	2404831.0	2458526.0	2513546.0	2566266.0
1179	Algeria	DZA	Population, total	SP.POP.TOTL	16149025.0	16607707.0	17085801.0	17582904.0	18102266.0
1243	Angola	AGO	Population, total	SP.POP.TOTL	6761380.0	7024000.0	7279509.0	7533735.0	7790707.0
1307	Argentina	ARG	Population, total	SP.POP.TOTL	25462302.0	25865776.0	26264681.0	26661398.0	27061047.0

Рис. 3.7. Первые несколько строк датафрейма `population_df`

5. Создайте динамическую переменную `year` для года и переменную `year_df` с датафреймом, содержащим колонки с названием страны и ука-

занным годом. После этого отсортируйте датафрейм по годам в порядке убывания и извлеките первые 20 строк:

```
year = '2010'
year_df = population_df[['Country Name', year]].sort_values(year,
ascending=False)[:20]
```

6. Поскольку в переменной `year_df` у нас хранится датафрейм с двумя колонками и сортировкой, мы можем легко построить столбчатую диаграмму, как мы делали ранее с точечной диаграммой. Также можем добавить динамический заголовок с использованием выбранного года, как показано ниже:

```
fig = go.Figure()
fig.add_bar(x=year_df['Country Name'],
y=year_df[year])
fig.layout.title = f'Top twenty countries by population - {year}'
fig.show()
```

Результатом будет изображение, показанное на рис. 3.8.

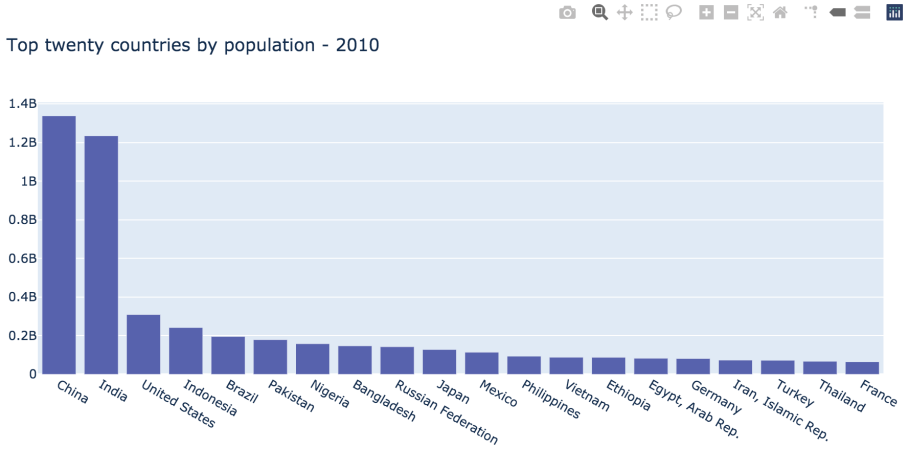


Рис. 3.8. Столбчатая диаграмма с 20 лидирующими странами по численности населения на 2010 год

Как видите, после получения нужного набора данных с правильной сортировкой можно всего с помощью нескольких строк кода построить диаграмму. Также заметьте, что по умолчанию значения по оси *y* выводятся в миллиардах, чтобы было легче читать график.

Здесь мы не вывели заголовки на оси – график как бы сам предполагает, что по горизонтальной оси у нас располагаются страны, а по вертикальной – численность населения в них.

Что касается года, то мы произвольно выбрали 2010-й, но на самом деле хотим, чтобы пользователь мог сам выбирать нужный ему год из присутствующих в наборе данных.

Давайте преобразуем фрагмент кода с выводом диаграммы в функцию, принимающую в качестве входного параметра год:

```
def plot_countries_by_population(year):
    year_df = ...
    fig = go.Figure()
    ...
    fig.show()
```

Эта функция также будет строить график, но для конкретного переданного ей года. Вы наверняка подумали, что эту функцию можно легко преобразовать в функцию обратного вызова, добавив всего одну строку кода. Именно это мы и сделаем, но сперва немного поговорим о преобразовании и подготовке данных, а также о том, как эти процессы влияют на их визуализацию. Просто данный пример прекрасно иллюстрирует эту тему.

Преобразование данных как важная часть процесса визуализации

Предыдущий пример состоял из шести шагов. При этом первые пять были направлены на преобразование исходных данных с извлечением двух массивов: для стран и численности населения, и только на последнем шаге мы занялись построением диаграммы. В результате мы написали гораздо больше кода для манипуляций с данными в сравнении с их визуализацией.

Если говорить об интеллектуальных усилиях, приложенных для построения столбчатой диаграммы, то их потребовалось не больше, чем для вывода точечной диаграммы в предыдущем примере, когда мы ограничились одной строкой кода `add_scatter(x=[1, 2, 3], y=[4, 2, 3])`. Здесь мы сделали то же самое, просто с другим методом и другими входными данными.

Что касается этапа подготовки данных, то здесь мы видим большие отличия в сравнении с построением точечной диаграммы. Во-первых, мы узнали, что в столбце **Country Name** хранятся данные не только о странах. Для этого нам пришлось провести визуальный анализ данных и исключить из списка стран регионы. Как мы могли узнать, что `"South Asia"` – это регион, а `"South Africa"` – страна? Мы могли просто знать это, а могли использовать список стран для проверки. Кроме того, мы обнаружили, что годы хранятся в виде строк, а не чисел. Я узнал это, когда получил ошибку `KeyError`, попытавшись обратиться к данным за конкретный год. Обычно процесс подготовки и преобразования исходных данных занимает достаточно много времени, зато визуализация после этого выполняется быстро и безболезненно.

В главе 4 мы посвятим процессу манипуляции с данными больше времени и представим несколько техник, которые могут оказаться очень полезными. Но вы должны понимать, что ваши навыки и умения в области преобразования данных, объединения наборов данных и использования регулярных выражений без применения не останутся, здесь им найдется применение. Также очень важное значение приобретает знание предметной области. В нашем случае, к примеру, мы должны были четко понимать разницу между странами и регио-

нами. По окончании подготовки данных вы можете применять различные техники и алгоритмы их анализа и визуализации, а также использовать методы машинного обучения, и для этого вам потребуется написать совсем немного строчек кода.

Итак, давайте попробуем сделать написанную ранее функцию интерактивной при помощи выпадающего списка и механизма обратного вызова.

Придание графику интерактивности за счет обратного вызова

Сначала мы реализуем свою задумку в JupyterLab, после чего перенесем код в наше приложение. В изолированном окружении наш атрибут `app.layout` будет содержать два компонента:

- выпадающий список (`Dropdown`): в нем будут присутствовать все годы, представленные в нашем наборе данных;
- график (`Graph`): это новый компонент, который мы раньше не применяли, но теперь будем использовать очень активно. Добавление компонента `Graph` в макет приведет к появлению пустой области диаграммы. Если помните, при изменении компонента в функции обратного вызова мы должны передать свойства `component_id` и `component_property`. В нашем примере мы будем менять свойство `figure`, принадлежащее компоненту `Graph`.

С секциями импорта и создания экземпляра приложения вы уже знакомы, так что мы акцентируем внимание исключительно на атрибуте `app.layout`:

```
app.layout = html.Div([
    dcc.Dropdown(id='year_dropdown',
                 value='2010',
                 options=[{'label': year, 'value': str(year)}
                          for year in range(1974, 2019)]),
    dcc.Graph(id='population_chart'),
])
```

Пока с компонентом `Graph` нет ничего особенного. Мы просто создали его наряду с компонентом `Dropdown` и присвоили ему описательный идентификатор.

Уверен, вы заметили, что мы немного изменили значения ключей `label` и `value` в списке `options` компонента `Dropdown`. А именно мы присвоили ключу `value` значение `str(year)`. Поскольку `options` представляет собой список словарей, созданный при помощи *генератора списков* (list comprehension), мы изначально получим список целочисленных значений. Выбранное значение будет использоваться для выбора столбца с таким заголовком. Но в наборе данных названия колонок представлены как строковые значения, а значит, обратиться к колонке следующим образом не получится: `population_df[2010]`. Столбца, названного таким числовым значением, в ней просто не найдется. В то же время

есть колонка со строковым заголовком **2010**. Таким образом, получив список числовых значений, мы обязаны преобразовать его в строки.

Также мы добавили новый параметр, который ранее не использовали. Параметр `value` компонента `Dropdown` отвечает за значение по умолчанию, которое увидит пользователь, впервые открыв приложение. Лучше всегда указывать такое значение, вместо того чтобы показывать пользователю пустой список.

В некоторых случаях вы можете действовать не так, как показано в этом примере. Например, вы можете оставить атрибут `value` как есть, но при этом изменить атрибут `label`. Допустим, если ваши данные введены в нижнем регистре, вы могли бы отобразить их в верхнем. В нашем примере с выбором цвета из предыдущей главы можно было бы написать следующий код:

```
dcc.Dropdown(options=[{'label': color.title(), 'value': color} for color in
['blue', 'green', 'yellow']])
```

С точки зрения функции обратного вызова значения цветов останутся прежними, поскольку она работает с параметром `value`. Но пользователь в выпадающем списке увидит значения цветов, написанные с большой буквы: `"Blue"`, `"Green"` и `"Yellow"`.

Запуск нашего приложения с двумя компонентами приведет к выводу, показанному на рис. 3.9.

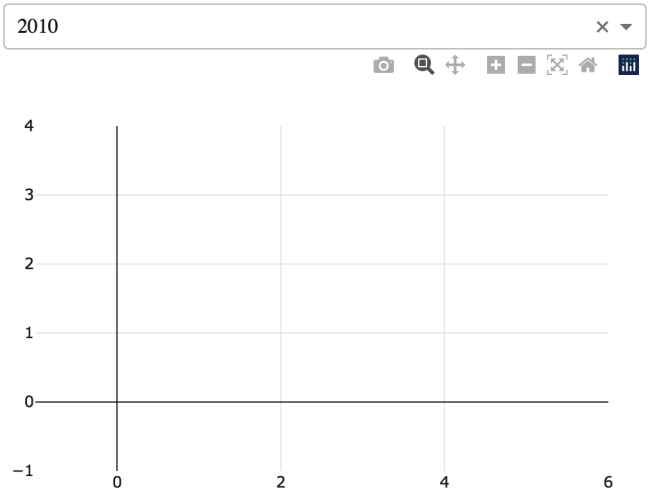


Рис. 3.9. Приложение с компонентом `Dropdown` со значением по умолчанию и пустым графиком

Мы уже написали функцию, принимающую на вход данные о годе и возвращающую столбчатую диаграмму с 20 странами, лидирующими по численности населения. Преобразование в функцию обратного вызова выполняется в одну строку, как показано ниже:

```
@app.callback(Output('population_chart', 'figure'),
              Input('year_dropdown', 'value'))
```

```
def plot_countries_by_population(year):
    year_df = ...
    fig = go.Figure()
    ...
    return fig
```

В нашей предыдущей функции последней строкой стояла инструкция `fig.show()`, а в функции обратного вызова мы используем объект `Figure` в качестве возвращаемого значения. Причина в том, что в первом случае мы запускали функцию интерактивно, без контекста приложения или обратного вызова. В данном же варианте у нас есть компонент с идентификатором `population_chart`, и, что более важно, нам нужно изменить его свойство `figure`. Возвращение объекта графика из функции обратного вызова позволит передать его компоненту `Graph` и изменить его атрибут `figure`.

Запуск этого приложения приведет к построению динамических графиков на основе выбора пользователя, что видно на рис. 3.10.

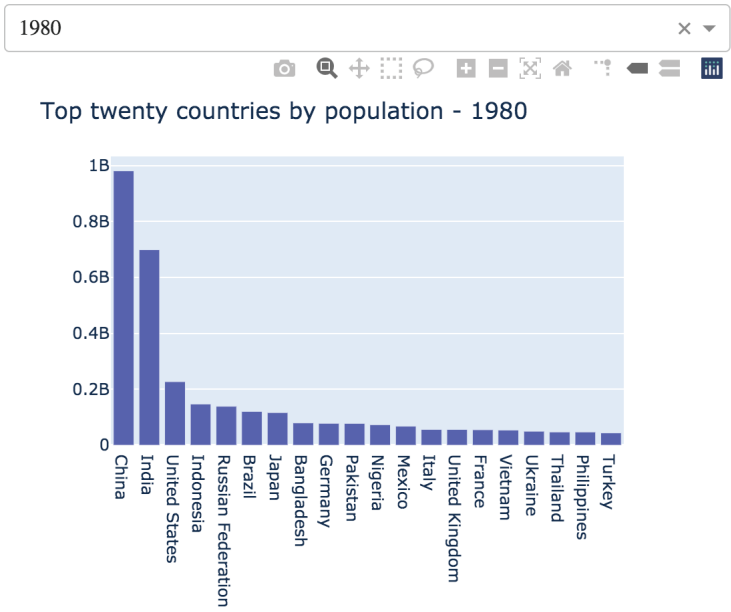


Рис. 3.10. Приложение со столбчатой диаграммой, построенной на основе пользовательского выбора

Если сравнить этот рисунок с рис. 3.8, можно заметить, что названия стран стали выводиться вертикально, тогда как раньше они располагались под углом. Причина в том, что первый график был выведен в окне браузера большего размера. Это еще одно удобство, которое предлагает нам Plotly без необходимости выполнять для этого какие-либо действия. Все ваши графики по умолчанию будут адаптироваться под размеры окна, что добавляет приложению гибкости. То же самое можно сказать про приложения, стилизованные с помощью пакета **Dash Bootstrap Components**.

Итак, мы создали изолированное приложение, работающее независимо. Теперь посмотрим, что нужно сделать, чтобы перенести этот функционал в наше исходное приложение.

Добавление функционала в приложение

Актуальная версия нашего приложения включает в себя компонент `Dropdown` и под ним элемент `Div` с отчетом о численности населения за 2010 год. Ниже располагается компонент `Tabs`. Давайте вставим новые компоненты `Dropdown` и `Graph` непосредственно под областью с отчетом и над компонентом `Tabs`. Также добавим функцию обратного вызова.

1. Скопируйте новые компоненты в атрибут `app.layout`:

```
...
html.Br(),
html.Div(id='report'),
html.Br(),
dcc.Dropdown(id='year_dropdown',
                value='2010',
                options=[{'label': year, 'value': str(year)}
                            for year in range(1974, 2019)]),
dcc.Graph(id='population_chart'),

dbc.Tabs([
...
```

2. Скопируйте определение функции обратного вызова и разместите его где-то после закрывающего тега верхнеуровневого элемента `Div` в составе атрибута `app.layout`:

```
@app.callback(Output('population_chart', 'figure'),
              Input('year_dropdown', 'value'))
def plot_countries_by_population(year):
  fig = go.Figure()
  year_df = population_df[['Country Name',
year]].sort_values(year, ascending=False)[:20]
  fig.add_bar(x=year_df['Country Name'], y=year_df[year])
  fig.layout.title = f'Top twenty countries by population - {year}'
  return fig
```

3. Добавьте определение списка `regions` и датафрейма `population_df` после строки с чтением данных из файла CSV в датафрейм `poverty_data`. Порядок здесь важен, поскольку датафрейм `population_df` строится на основе списка `regions` и датафрейма `poverty_data`. Ниже показан правильный порядок этих определений:

```
poverty_data = ...
regions = ...
population_df = ...
```

Если теперь запустить приложение, вы увидите вывод, показанный на рис. 3.11.

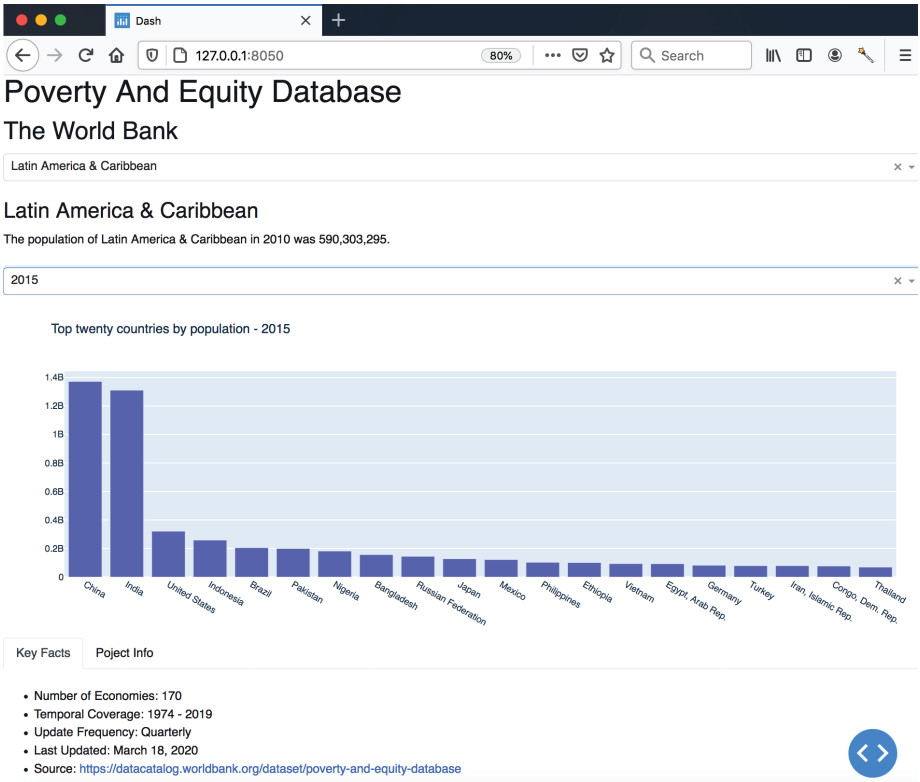


Рис. 3.11. Приложение с добавленными компонентами `Dropdown` и `Graph`

Если открыть окно отладчика и нажать на кнопку **Callbacks**, вы увидите обновленный вид обратных вызовов с названиями компонентов, с которыми они связаны (с идентификаторами и свойствами), показанный на рис. 3.12.

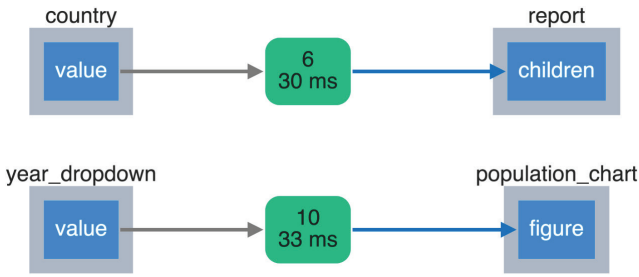


Рис. 3.12. Обратные вызовы приложения в визуальном отладчике

Теперь наше приложение показывает больше информации и позволяет пользователю извлекать данные из набора динамически. Мы определили две функции обратного вызова и макет, содержащий несколько компонентов различных типов. Также мы написали порядка 90 строк кода. Пока у вас не слишком много компонентов на макете, вы можете просто добавлять их в код и ни о чем не заботиться. Но с увеличением масштаба приложения придется осваивать различные техники организации кода и его рефакторинга.

Давайте подытожим обсуждение объекта `Figure` простой и понятной темой, не требующей написания большого количества кода, после чего пройдемся по аспектам, которые мы изучили.

Создание тем для графиков

Создание тем для графиков, в отличие от приложений, может показаться очень интересным занятием и способно сэкономить немало времени при необходимости менять их. Доступ к теме графика осуществляется при помощи атрибута `template` объекта `layout`, как показано ниже:

```
fig.layout.template = template_name
```

На рис. 3.13 показаны четыре разные темы с их названиями.

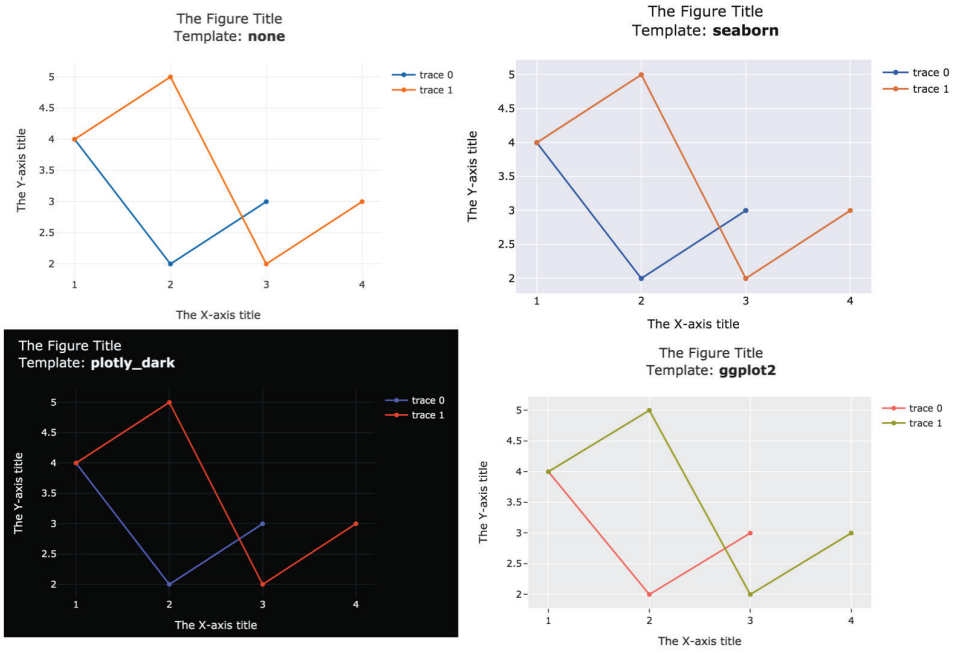


Рис. 3.13. Четыре темы графиков

Полный список доступных тем можно посмотреть в конфигурационном объекте `plotly.io.templates`.

Всегда лучше тщательно подбирать темы для объектов, чтобы они были совместимы с темой всего приложения. Также вы можете использовать эту

библиотеку в качестве отправной точки и изменять нужные вам элементы в теме по желанию.

Давайте вспомним, что мы узнали из этой главы.

Заключение

Начали главу мы со знакомства с объектом `Figure`, его компонентами и их атрибутами. Мы узнали, как можно создавать эти объекты шаг за шагом и менять нужные нам свойства. Мы также обратили пристальное внимание на два главных атрибута объекта `Figure`: `data` и `layout`. Кроме того, рассмотрели разные варианты конвертации графиков, а затем создали диаграмму на основе нашего набора данных и внедрили ее в приложение.

На данном этапе вы уже знаете, как создавать и структурировать приложения Dash, как добавить им интерактивности при помощи создания функций обратного вызова, связывающих разные компоненты воедино, и как строить диаграммы внутри приложений.

Кроме того, вы узнали о том, как управлять разными свойствами и характеристиками созданных графиков, чтобы их чтение не было затруднено, а пользователь мог больше времени уделить не пониманию того, что видит, а непосредственно анализу.

До сих пор мы лишь вскользь касались вопросов преобразования и подготовки данных для визуализации, а теперь пришло время поговорить об этом более подробно. В следующей главе мы познакомимся с библиотекой **Plotly Express**, позволяющей легко и непринужденно создавать сложные графики и диаграммы.

Глава 4

Подготовка и преобразование данных. Введение в Plotly Express

Как вы уже поняли, процесс подготовки и преобразования исходных данных зачастую может занимать куда больше времени, чем их визуализация. Иными словами, если вы потратите значительные усилия на подготовку данных и приведение их в вид, пригодный для анализа, вывод их на графике не составит труда. До сих пор мы использовали лишь малую часть нашего набора данных и практически не делали никаких преобразований и не меняли формат. А при построении графиков мы создавали их с нуля, после чего добавляли слои, ряды данных, заголовки и т. д.

В этой главе мы получше познакомимся с нашим набором данных и попробуем привести его в интуитивный и простой для анализа вид. Это позволит нам применить новый подход к созданию визуализаций – при помощи библиотеки **Plotly Express**. Вместо того чтобы начинать с пустого прямоугольника и постепенно добавлять на него слои, мы возьмем за основу признаки (столбцы) нашего набора данных и будем строить визуализацию на них. Иначе говоря, станем больше ориентироваться на сами данные, а не на их вывод. Попутно мы сравним два этих подхода и обсудим, когда и какой из них лучше использовать.

В этой главе мы рассмотрим следующие темы:

- длинный формат данных (tidy);
- роль навыков в области преобразования данных;
- знакомство с Plotly Express.

Технические требования

В этой главе мы фактически не будем использовать какие-то новые пакеты, но таковым можно считать объемный модуль Plotly Express из состава Plotly. Также мы будем активно использовать библиотеку `pandas` для подготовки, форматирования и манипуляций с данными. Главным образом мы будем работать в среде JupyterLab, а в качестве наборов данных станем использовать файлы из папки `data`, располагающейся в корне репозитория.

Исходный код этой главы можно скачать в репозитории GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_04**.

Сопроводительные видеофрагменты к главе можно посмотреть по адресу **https://bit.ly/3suvKi4**.

Давайте начнем с рассмотрения различных форматов данных и особенностей их использования.

Длинный формат данных (tidy)

В данной главе мы будем работать с достаточно сложным набором данных. Он состоит из четырех файлов CSV и содержит информацию почти обо всех странах и регионах в мире. В нашем наборе данных присутствует более 60 показателей (метрик) за более чем 40 лет, что дает возможность сочетать и анализировать их самыми разными способами.

Перед тем как приступить к процессу преобразования данных, я бы хотел продемонстрировать вам нашу конечную цель на примере, чтобы вы понимали, к чему мы будем стремиться. Вы также должны понять, почему мы готовы тратить столько времени на достижение этих изменений.

Примеры графиков Plotly Express

Библиотека Plotly Express поставляется с несколькими наборами данных, которые можно использовать для обучения и проверки своих гипотез. Они находятся в модуле `data` внутри пакета `plotly.express` и вызываются как функции, возвращающие соответствующий набор данных. Давайте взглянем на популярный набор данных Gapminder, написав следующий код:

```
import plotly.express as px
gapminder = px.data.gapminder()
gapminder
```

Запуск этого кода приведет к выводу на экран датафрейма `gapminder`, как показано на рис. 4.1.

	country	continent	year	lifeExp	pop	gdpPercap	iso_alpha	iso_num
0	Afghanistan	Asia	1952	28.801	8425333	779.445314	AFG	4
1	Afghanistan	Asia	1957	30.332	9240934	820.853030	AFG	4
2	Afghanistan	Asia	1962	31.997	10267083	853.100710	AFG	4
3	Afghanistan	Asia	1967	34.020	11537966	836.197138	AFG	4
4	Afghanistan	Asia	1972	36.088	13079460	739.981106	AFG	4
...	...	...	...	...	...	...	...	...
1699	Zimbabwe	Africa	1987	62.351	9216418	706.157306	ZWE	716
1700	Zimbabwe	Africa	1992	60.377	10704340	693.420786	ZWE	716
1701	Zimbabwe	Africa	1997	46.809	11404948	792.449960	ZWE	716
1702	Zimbabwe	Africa	2002	39.989	11926563	672.038623	ZWE	716
1703	Zimbabwe	Africa	2007	43.487	12311143	469.709298	ZWE	716

1704 rows × 8 columns

Рис. 4.1. Набор данных Gapminder из состава Plotly Express

Структура набора данных довольно простая. Для каждого уникального набора и страны (поле **country**), континента (поле **continent**) и года (поле **year**) у нас есть три метрики: **lifeExp**, **pop** и **gdpPercap**. В полях **iso_alpha** и **iso_num** хранятся закодированные значения для стран.

Посмотрим, как можно агрегировать данные в датафрейме `gapminder` при помощи точечной диаграммы.

На ось *x* мы можем вынести значения показателя **gdpPercap**, а в качестве функции от них на ось *y* поместим метрику **lifeExp**, чтобы узнать зависимость между ними. Неплохо было бы также с помощью размера маркеров отображать численность населения стран.

Кроме того, мы можем разделить визуализацию по горизонтали на подграфики или *фасеты* (facet) с помощью параметра `facet_col`, по одному для каждого континента, и отобразить это при помощи заголовков. Также маркерам для каждого континента можно задать собственный цвет, а для большей ясности снабдить график заголовком *«Life Expectancy and GDP per capita. 1952–2007»* (Средняя продолжительность жизни и валовой внутренний продукт на душу населения. 1952–2007).

Чтобы пользователь понимал, о чем идет речь на графике, мы подпишем оси, преобразовав имя показателя `gdpPercap` в *GDP per Capita* (валовой внутренний продукт на душу населения), а `lifeExp` – в *Life Expectancy* (средняя продолжительность жизни).

Вполне уместно ожидать, что подобный график будет характеризоваться выбросами и не будет распределен нормально, в связи с чем лучше будет установить для оси *x* логарифмическую шкалу (`log_x`). Диапазон значений по оси *y* (`range_y`) должен соответствовать интервалу [20, 100], чтобы можно было видеть, как варьируется показатель средней продолжительности жизни в рамках фиксированного диапазона.

При наведении мышью на маркеры должна отображаться подробная информация о стране, а заголовок всплывающего окна (`hover_name`) должен соответствовать названию выбранной страны. Если информацию по всем годам разместить поверх друг друга, график окажется совершенно нечитаемым. Предлагаю воспользоваться анимацией (`animation_frame`) для каждого отдельного года.

Было бы здорово, если бы у нас была кнопка запуска анимации, с помощью которой можно было бы двигаться по годам, как в видео, с возможностью сделать паузу на любом периоде.

Высоту для нашего графика (`height`) установим на отметке 600 пикселей:

```
px.scatter(data_frame=gapminder,
           x='gdpPercap',
           y='lifeExp',
           size='pop',
           facet_col='continent',
           color='continent',
           title='Life Expectancy and GDP per capita. 1952 - 2007',
           labels={'gdpPercap': 'GDP per Capita', 'lifeExp': 'Life Expectancy'},
```

```
            log_x=True,
            range_y=[20, 100],
            hover_name='country',
            animation_frame='year',
            height=600,
            size_max=90)
```

Запуск этого кода приведет к отображению визуализации, показанной на рис. 4.2.

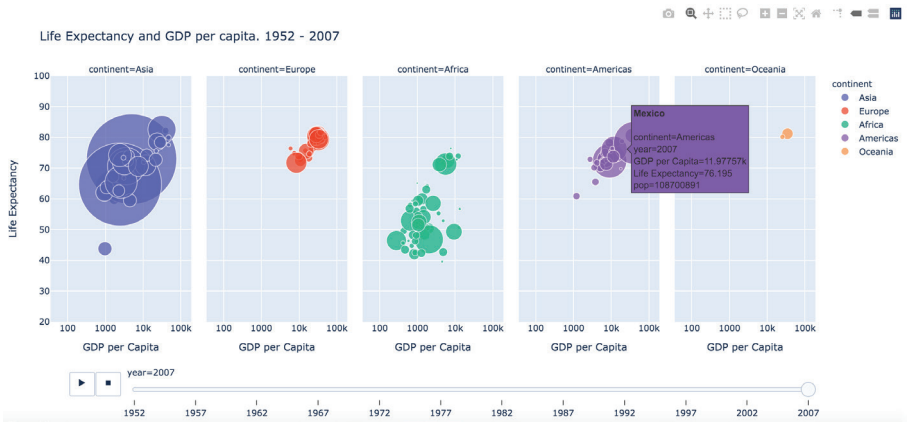


Рис. 4.2. Интерактивный график на основе набора данных Gapminder при помощи Plotly Express

Как видите, в тексте, описывающем построение этой диаграммы, гораздо больше букв, чем в самом коде. По сути, эта визуализация была создана при помощи всего одной строчки кода.

Нажатие на кнопку запуска анимации в левом нижнем углу приведет к изменениям на графиках – для каждого года вы будете видеть свои показатели. При желании вы можете поставить анимацию на паузу или переместить ползунок на любой нужный вам год. Таким образом, вы можете наблюдать за тем, как с годами менялась взаимосвязь между отображаемыми показателями.

Если навести мышью на точку данных на любом из представленных графиков, вы увидите всплывающее окно с подробной информацией о стране и ее показателях в отчетном году. Аргументу `hover_name` мы присвоили значение `'country'`, и именно поэтому название страны отображается во всплывающем окне жирным шрифтом в виде заголовка.

Как видно на графиках, наши маркеры по странам по большей части накладываются друг на друга, что затрудняет чтение данных. Но графики Plotly интерактивны по своей природе, так что вы можете использовать панель инструментов для приближения и отдаления, а также выбора произвольной прямоугольной области, которую необходимо увеличить.

На рис. 4.3 показано, каким станет внешний вид графика, если выделить прямоугольником все точки данных на африканском континенте. Насколько легче стало их анализировать.

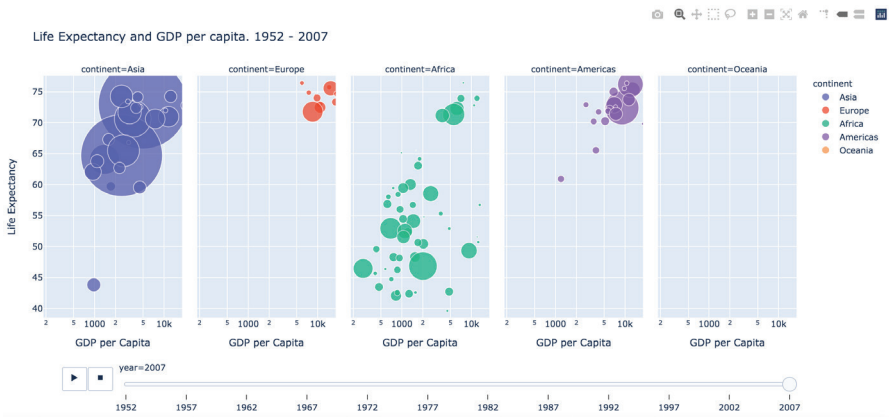


Рис. 4.3. Увеличение выбранной области на графике

Заметьте, что остальные графики также были увеличены до размера выбранной области. Вы можете самостоятельно исследовать все доступные опции интерактивного просмотра графиков, но, полагаю, вы видите, насколько мощным и интуитивным инструментом является Plotly Express.

Существуют две главные причины, позволившие нам построить столь гибкий график. Во-первых, инструмент Plotly Express изначально построен таким образом, чтобы мощные визуализации можно было создавать при помощи минимального вмешательства программиста. Об этой особенности данной библиотеки мы еще поговорим подробнее далее. Во-вторых, важную роль в процессе построения графиков при помощи Plotly Express играет структура данных. Если данные приведены в вид, удобный для анализа, визуализировать их будет очень просто.

Основные атрибуты длинного формата данных (tidy)

Одной из ключевых особенностей удобной структуры данных является возможность представить любой маркер на графике в виде отдельно взятой независимой строки. Каждое значение в строке принадлежит отдельной колонке. В результате каждый столбец описывает свою переменную, представленную определенным типом данных. Это значительно облегчает установку цвета точек данных, их размера и других атрибутов – мы просто объявляем, какой столбец каким визуальным атрибутом должен быть выражен.

Обратите внимание, что сказанное мной очень походит на определение датафрейма, характеризующегося следующими свойствами:

- набором столбцов, каждый из которых содержит значения определенного типа;
- независимостью типов данных разных столбцов внутри набора данных;
- равными размерами столбцов, хотя в каждом из них могут присутствовать пропущенные значения.

С точки зрения концепции главным отличием длинного формата датафрейма от обычного является то, что в нем одному наблюдению строго соответствует

одна строка датафрейма (страна, человек, бренд или их сочетание), а переменной – один столбец (численность населения, размер, длина, высота, прибыль и т. д.). К примеру, в столбце **country** могут содержаться названия стран и ничего больше. К тому же страны не должны присутствовать и в других колонках, чем исключается появление неоднозначности в данных.

Этот формат датафрейма не является единственно правильным или более верным по сравнению с остальными. Он просто наиболее интуитивно понятный, и с ним легче всего работать. Требованиями для построения визуализации, показанной ранее, были наличие в датафрейме набора значений для оси *x* и такого же по длине набора значений для оси *y*. Для других признаков на диаграмме, таких как цвет или размер маркеров, нам также понадобятся наборы значений такой же длины, чтобы можно было поставить им в соответствие наши точки данных. И датафрейм как нельзя лучше подходит под перечисленные требования.

Чтобы на построенном нами макете все точки данных стали одного размера, достаточно удалить аргумент `size`. Изменение аргумента `facet_col` на `facet_row` приведет к тому, что отдельные графики будут располагаться поверх друг друга стопочкой, а не бок о бок. Таким образом, незначительные корректировки кода способны привести к кардинальным изменениям на диаграмме. Это не сложнее, чем щелкать переключателями на дашборде!

Полагаю, конечная цель вам ясна. Нам необходимо рассмотреть наши файлы CSV на предмет возможности их преобразования в длинный формат. В результате в каждом столбце должна храниться информация о конкретной переменной (год, численность населения, *индекс Джини* (Gini index) и т. д.), а в каждой строке – определенное наблюдение (комбинация страны, года, показателя и других значений). После приведения данных в нужный нам вид мы сможем взглянуть на них, проанализировать, выделить значимые для нас показатели и визуализировать с помощью функции пакета Plotly Express.

Процесс покажется вам простым и понятным, стоит только начать. Так давайте приступим!

Роль навыков в области преобразования данных

На практике мы очень редко получаем исходные данные в виде, пригодном для анализа. Обычно они представлены в виде нескольких файлов, которые нам необходимо объединить, а часто данные также нуждаются в нормализации и очистке. Именно поэтому навыки в области манипулирования и преобразования данных всегда будут востребованы на этапе их подготовки к визуализации. И в этой главе мы будем подробно говорить об этих аспектах.

План подготовки данных включает в себя следующие шаги.

1. Исследование всех исходных файлов.
2. Проверка наличия в данных нужных сведений и анализ типов данных, в которых они представлены.
3. Трансформация данных при необходимости.

4. Объединение различных датафреймов для расширения числа способов описания данных.

Пройдемся по этим стадиям.

Исследование исходных файлов

Начнем с чтения нужных файлов, располагающихся в папке `data`:

```
import os
import pandas as pd
pd.options.display.max_columns = None
os.listdir('data')
['PovStats**Series**.csv',
 'PovStats**Country**.csv',
 'PovStats**Country-Series**.csv',
 'PovStats**Data**.csv',
 'PovStats**FootNote**.csv']
```

Для ясности мы будем отличающуюся часть имен файлов брать в качестве имен переменных для соответствующих датафреймов: `'PovStats<name>.csv'`.

Файл Series

Начнем с загрузки файла *PovStatsSeries.csv* с использованием следующего кода:

```
series = pd.DataFrame('data/'PovStatsSeries.csv')
print(series.shape)
series.head()
```

В результате будет выведена размерность датафрейма (количество строк и столбцов), а также первые пять строк из него, как показано на рис. 4.4.

```
(64, 21)
```

	Series Code	Topic	Indicator Name	Short definition	Long definition	Unit of measure	Periodicity	Base Period	Other notes	Aggregation method	Limitations and exceptions	Notes from original source	General comments
0	SI.DST.02ND.20	Poverty: Income distribution	Income share held by second 20%	NaN	Percentage share of income or consumption is t...	%	Annual	NaN	NaN	NaN	Despite progress in the last decade, the chall...	NaN	The World Bank's internationally comparable po...
1	SI.DST.03RD.20	Poverty: Income distribution	Income share held by third 20%	NaN	Percentage share of income or consumption is t...	%	Annual	NaN	NaN	NaN	Despite progress in the last decade, the chall...	NaN	The World Bank's internationally comparable po...
2	SI.DST.04TH.20	Poverty: Income distribution	Income share held by fourth 20%	NaN	Percentage share of income or consumption is t...	%	Annual	NaN	NaN	NaN	Despite progress in the last decade, the chall...	NaN	The World Bank's internationally comparable po...
3	SI.DST.05TH.20	Poverty: Income distribution	Income share held by highest 20%	NaN	Percentage share of income or consumption is t...	%	Annual	NaN	NaN	NaN	Despite progress in the last decade, the chall...	NaN	The World Bank's internationally comparable po...
4	SI.DST.10TH.10	Poverty: Income distribution	Income share held by highest 10%	NaN	Percentage share of income or consumption is t...	%	Annual	NaN	NaN	NaN	Despite progress in the last decade, the chall...	NaN	The World Bank's internationally comparable po...

Рис. 4.4. Размерность и первые пять строк датафрейма PovStatsSeries

Похоже, мы имеем дело с 64 индикаторами, для каждого из которых указан 21 атрибут или описание. Данные уже представлены в длинном формате – в каждом столбце содержится ровно один атрибут, а в каждой строке представлена вся информация о конкретном индикаторе. Так что здесь нам просто нечего менять. Достаточно рассмотреть данные и удостовериться, что с ними все в порядке.

Информации в этом файле достаточно, чтобы построить некий служебный дашборд для индикаторов и разместить его на отдельной странице. Данных в каждой строке, кажется, хватает, чтобы создать отдельную страницу с заголовком, описанием и подробностями каждого индикатора. Основная область страницы может вместить в себя визуализацию индикатора по всем странам и годам. Это одна из идей.

Давайте внимательнее присмотримся к некоторым наиболее интересующим нас колонкам:

```
series['Topic'].value_counts()
Poverty: Poverty rates           45
Poverty: Shared prosperity       10
Poverty: Income distribution      8
Health: Population: Structure     1
Name: Topic, dtype: int64
```

Как видите, все индикаторы распределены на четыре группы по столбцу `Topic`. Общее число индикаторов, относящихся к каждой группе, показано на выводе выше.

Также интересно посмотреть, что у нас находится в столбце `Unit of measure`:

```
series['Unit of measure'].value_counts(dropna=False)
%             39
NaN           22
2011 PPP $     3
Name: Unit of measure, dtype: int64
```

Как видите, в основном у нас индикаторы представлены либо в процентах, либо в неизвестных величинах (NaN). Это может помочь нам позже при группировании определенных типов графиков.

Еще одна важная для нас колонка в этом наборе данных – это **Limitations and exceptions**. Обычно подобные метаданные предоставляют ценную информацию о возможных ошибках в исходных сведениях и дают понять, что нужно учитывать при анализе данных. Столбцы с ограничениями желательно внимательно просматривать и анализировать на предмет дублирования или вхождения в какие-то группы. В следующем фрагменте кода мы сгруппируем датафрейм `series` по столбцу `Topic`, после чего подсчитаем количество значений в поле `Limitations and Exceptions`, а также выведем количество уникальных значений:

```
(series
 .groupby('Topic')
 ['Limitations and exceptions']
 .agg(['count', pd.Series.nunique])
 .style.set_caption('Limitations and Exceptions'))
```

Результат показан на рис. 4.5.

Limitations and Exceptions

Topic	count	nunique
Health: Population: Structure	1	1
Poverty: Income distribution	8	2
Poverty: Poverty rates	25	3
Poverty: Shared prosperity	4	2

Рис. 4.5. Подсчет значений в поле **Limitations and Exceptions**

Возможно, эта информация будет хорошим ориентиром для нас при анализе различных индикаторов. Это также поможет нашим пользователям, которые будут понимать, что именно они анализируют.

Файл Country

Теперь взглянем на содержимое файла *PovStatsCountry.csv*:

```
country =\
pd.read_csv('data/PovStatsCountry.csv',na_values='',
                      keep_default_na=False)
print(country.shape)
country.head()
```

Здесь мы также увидим размерность датафрейма и первые пять строк содержимого, как показано на рис. 4.6.

```
(184, 31)
```

	Country Code	Short Name	Table Name	Long Name	2-alpha code	Currency Unit	Special Notes	Region	Income Group	WB-2 code	National accounts base year	National accounts reference year	SNA price valuation	Lending category	Other groups	System of National Accounts	Alternative conversion factor
0	AFG	Afghanistan	Afghanistan	Islamic State of Afghanistan	AF	Afghan afghani	NaN	South Asia	Low income	AF	2002/03	NaN	Value added at basic prices (VAB)	IDA	HIPC	Country uses the 1993 System of National Accou...	NaN
1	AGO	Angola	Angola	People's Republic of Angola	AO	Angolan kwanza	NaN	Sub-Saharan Africa	Lower middle income	AO	2002	NaN	Value added at basic prices (VAB)	IBRD	NaN	Country uses the 1993 System of National Accou...	1991–96
2	ALB	Albania	Albania	Republic of Albania	AL	Albanian lek	NaN	Europe & Central Asia	Upper middle income	AL	Original chained constant price data are resca...	2010	Value added at basic prices (VAB)	IBRD	NaN	Country uses the 2008 System of National Accou...	NaN
3	ARG	Argentina	Argentina	Argentine Republic	AR	Argentine peso	NaN	Latin America & Caribbean	Upper middle income	AR	2004	NaN	Value added at basic prices (VAB)	IBRD	NaN	Country uses the 2008 System of National Accou...	1971–84; 2012–15
4	ARM	Armenia	Armenia	Republic of Armenia	AM	Armenian dram	NaN	Europe & Central Asia	Upper middle income	AM	Original chained constant price data are resca...	2012	Value added at basic prices (VAB)	IBRD	NaN	Country uses the 2008 System of National Accou...	1990–95

Рис. 4.6. Фрагмент файла с информацией о странах

На этот раз при чтении из файла при помощи метода `read_csv` мы указали аргументы `keep_default_na=False` и `na_values=''`. Дело в том, что `pandas` интерпретирует строки вроде `NA` и `NaN` как отсутствующие значения. А у страны Намибия двухсимвольный код как раз `NA`, и мы бы не хотели потерять информацию о ней. Именно поэтому мы внесли такие корректировки в метод чтения данных. Это отличный пример того, как можно совершенно неожиданно потерять данные при чтении.

В открытом файле содержится важная информация о странах и регионах. Файл довольно небольшой, но он может быть нам очень полезен при фильтрации и группировке данных о странах. Здесь информация также представлена в длинном формате. Давайте посмотрим, какие колонки нас интересуют особенно.

Разумеется, это колонка **Region**. Можно посмотреть, какие регионы представлены в нашем наборе данных, и посчитать количество стран в регионах следующим образом:

```
country['Region'].value_counts(dropna=False).to_frame().style.
background_gradient('cividis')
```

Результат показан на рис. 4.7.

	Region
Europe & Central Asia	49
Sub-Saharan Africa	47
Latin America & Caribbean	25
East Asia & Pacific	24
nan	15
Middle East & North Africa	14
South Asia	8
North America	2

Рис. 4.7. Количество стран по регионам

Еще один столбец, который может быть нам интересен, – это **Income Group**. Если значения здесь проставлены верно, мы можем в будущем размножать графики по уровню дохода, как мы ранее делали с континентами:

```
country['Income Group'].value_counts(dropna=False)
Upper middle income    52
Lower middle income    47
High income            41
Low income             29
NaN                    15
Name: Income Group, dtype: int64
```

Наличие 15 значений NaN согласуется с объединенным количеством регионов и классификаций, что мы вскоре увидим. Уровень доходов стран не зависит от их географического положения.

Если вы взглянете на столбец **Short Name**, то увидите, что не все значения в нем описывают страны. Помимо них, есть географические регионы, такие как **Middle East & North Africa**, а также классификации, например **Lower middle income**. Между ними важно делать различие, и мы можем для этого создать дополнительную колонку, с помощью которой будем отличать страны от прочих значений.

Столбец **Region** показывает, какому региону принадлежит образование, описанное в колонке **Short Name**. Для регионов этот столбец будет содержать отсутствующее значение (NaN). Мы можем использовать это как фактор, на основании которого создадим логический столбец `is_country`:

```
country['is_country'] = country['Region'].notna()
```

На рис. 4.8 представлен фрагмент датафрейма, в котором присутствуют страны, регионы и классификации.

	Short Name	Region	is_country
170	Upper middle income	NaN	False
42	East Asia & Pacific	NaN	False
155	Syrian Arab Republic	Middle East & North Africa	True
84	Kyrgyz Republic	Europe & Central Asia	True
40	Dominican Republic	Latin America & Caribbean	True
21	Central African Republic	Sub-Saharan Africa	True
105	Middle East & North Africa	NaN	False
150	Slovak Republic	Europe & Central Asia	True
95	Lower middle income	NaN	False
28	Dem. Rep. Congo	Sub-Saharan Africa	True

Рис. 4.8. Данные с дополнительным столбцом `is_country`

Полный список этих категорий можно получить путем извлечения подмножества из датафрейма `country`, в котором колонка **Region** содержит отсутствующие значения. Выведем содержимое столбца **Short Name**:

```
country[country['Region'].isna()]['Short Name']
37     IDA countries classified as fragile situations
42                                East Asia & Pacific
43                              Europe & Central Asia
50           Fragile and conflict affected situations
70                                          IDA total
```

```
92                          Latin America & Caribbean
93                                         Low income
95                                Lower middle income
96                                Low & middle income
105                       Middle East & North Africa
107                                  Middle income
139                                     South Asia
147                               Sub-Saharan Africa
170                               Upper middle income
177                                              World
Name: Short Name, dtype: object
```

Этот процесс очень важен при планировании ваших дашбордов и приложений. Например, зная о том, что в нашем датафрейме содержится всего четыре классификации по уровню дохода, мы можем принять решение о выводе соответствующих графиков бок о бок. Если бы их было 20, такой вывод не имел бы смысла.

Давайте создадим еще один столбец, после чего перейдем к следующему файлу.

Поскольку мы здесь имеем дело со странами, мы можем использовать в качестве их идентификаторов национальные флаги. Хранятся флаги в виде эмодзи и представляют собой символы Unicode, а значит, мы можем выводить их в виде текста на наших графиках подобно любым другим текстовым элементам. Позже мы можем также рассмотреть возможность использования в наших визуализациях других символов – например, эмодзи со стрелками вверх и вниз, характеризующих подъемы и спады. Это бывает также полезно при наличии ограниченного места для вывода визуализации. Порой один эмодзи стоит тысячи слов!

Здесь важно отметить, что эмодзи национальных флагов кодируются при помощи конкатенации двух специальных символов, называемых *региональными индикаторными символами* (regional indicator symbol). К примеру, региональные символы для букв *A* и *B* выглядят так: 🇦 и 🇧.

Все, что нам нужно, – это получить двухбуквенный индикатор конкретной страны и по нему найти нужные символы, воспользовавшись функцией `lookup` из пакета `unicodedata`, входящего в состав *стандартной библиотеки Python* (Python Standard Library). Функция `lookup` принимает на вход название символа и возвращает сам искомый символ следующим образом:

```
from unicodedata import lookup
lookup('LATIN CAPITAL LETTER E')
'🇪'
lookup("REGIONAL INDICATOR SYMBOL LETTER A")
'🇦'
```

Получив две буквы, представляющие конкретную страну, мы можем найти соответствующие им символы и соединить их для получения нужного нам

флага. Давайте напишем для этого простую функцию. При этом нам нужно отловить ситуации, когда в качестве буквенных символов было передано значение NaN или сочетание букв не входит в список стран.

Для этого мы создадим переменную `country_codes`, которую будем использовать в качестве критерия. Если переданные символы не входят в этот список, вернем пустую строку, в противном случае – символ эмодзи, соответствующий флагу:

```
country_codes = country[country['is_country']]['2-alpha code'].dropna().
str.lower().tolist()
```

Теперь можно написать определение функции `flag`:

```
def flag(letters):
    if pd.isna(letters) or (letters.lower() not in country_codes):
        return ''
    L0 = lookup(f'REGIONAL INDICATOR SYMBOL LETTER {letters[0]}')
    L1 = lookup(f'REGIONAL INDICATOR SYMBOL LETTER {letters[1]}')
    return L0 + L1
```

С помощью этой функции мы можем создать и заполнить новый столбец с именем `flag`:

```
country['flag'] = [flag(code) for code in country['2-alpha code']]
```

На рис. 4.9 показан случайный набор стран, их флаги и содержимое столбца `is_country`.

	Short Name	flag	is_country
173	Uzbekistan		True
83	Kenya		True
130	Palau		True
182	Zambia		True
31	Comoros		True
102	Moldova		True
180	Yemen		True
170	Upper middle income		False
134	Paraguay		True
155	Syrian Arab Republic		True

Рис. 4.9. Строки со странами и их флагами

В строках, где в столбце **Short Name** стоит не страна, флаг выводится пустой. При этом мы выводим именно пустую строку, а не значение `NaN`, поскольку в дальнейшем нам может понадобиться конкатенировать названия стран с их флагами для заголовков графиков или иных целей, и с пустой строкой в этих случаях точно не возникнет проблем. Заметьте, что при сохранении датафрейма в файл и его повторном открытии `pandas` будет интерпретировать пустые строки как `NaN`, и вам придется либо конвертировать их, либо предотвратить такую интерпретацию.

Файл country-series

Наш следующий подопытный файл с именем *PovStatsCountry-Series.csv* содержит коды стран и описание источников данных об их численности населения. Рассмотрим этот файл внимательнее, если/когда поймем, что сможем использовать его в качестве источника метаданных на наших графиках.

Файл footnotes

Далее на очереди файл *PovStatsFootNote.csv*. Здесь мы видим пустую колонку с именем **Unnamed: 4**, от которой нам необходимо избавиться. Также нужно преобразовать строковое представление годов в числовое, убрав два первых символа `YR`. К примеру, значение `YR2015` должно быть преобразовано в число `2015`. В заключение переименуем колонки в полном соответствии с нашим датафреймом `series`, чтобы при необходимости их было легче объединять друг с другом:

```
footnote = pd.read_csv('data/PovStatsFootNote.csv')
footnote = footnote.drop('Unnamed: 4', axis=1)
footnote['Year'] = footnote['Year'].str[2:].astype(int)
footnote.columns = ['Country Code', 'Series Code', 'year', 'footnote']
footnote
```

На рис. 4.10 показано несколько строк из обработанного датафрейма `footnote`.

	Country Code	Series Code	year	footnote
0	AFG	SI.POV.NAHC	2007	Source: National Statictis and Information Aut...
1	AFG	SI.POV.NAHC	2011	Source: National Statictis and Information Aut...
2	AFG	SI.POV.NAHC	2016	Source: National Statictis and Information Aut...
3	AFG	SI.POV.NAHC.NC	2007	Source: National Statictis and Information Aut...
4	AFG	SI.POV.NAHC.NC	2011	Source: National Statictis and Information Aut...
...	...	...	...	...
33472	ZWE	SI.POV.NAHC.NC	2011	Source: PICES 2011/12, Poverty and Poverty Dat...
33473	ZWE	SI.POV.NOP1	2011	Estimated from grouped consumption data.
33474	ZWE	SI.POV.UMIC	2011	Estimated from grouped consumption data.
33475	ZWE	SI.POV.UMIC.GP	2011	Estimated from grouped consumption data.
33476	ZWE	SI.POV.UMIC.NO	2011	Estimated from grouped consumption data.

33477 rows × 4 columns

Рис. 4.10. Фрагмент датафрейма `footnote`

Как видите, у нас есть довольно много *примечаний* (footnote) об имеющихся данных, и нужно как-то интегрировать их с прочей информацией, чтобы пользователи видели общую картину происходящего. Похоже, что примечания базируются на сочетании столбцов со страной, индикатором и годом. Судя по тому, что данные во всех трех столбцах содержатся в том же виде, что и в других таблицах, должно быть несложно объединить их для дальнейшего использования.

Файл data

Теперь рассмотрим основной файл с данными, с которым мы уже работали в предыдущих главах, – *PovStatsData.csv*. На этот раз мы трансформируем его и объединим с другими датафреймами для получения всеобъемлющей картины.

Посмотрим, что у нас содержится в файле:

```
data = pd.read_csv('data/PovStatsData.csv')
data = data.drop('Unnamed: 50', axis=1)
print(data.shape)
data.sample(3)
```

Здесь мы удалили колонку с именем **Unnamed: 50** и вывели на экран размерность датафрейма и три случайно выбранные строки, как показано на рис. 4.11.

```
(11840, 50)
```

	Country Name	Country Code	Indicator Name	Indicator Code	1974	1975	1976	1977	1978	1979	1980
11657	Yemen, Rep.	YEM	Income share held by fourth 20%	SI.DST.04TH.20	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2970	Colombia	COL	Number of poor at $5.50 a day (2011 PPP) (mill...	SI.POV.UMIC.NO	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1223	Angola	AGO	Growth component of change in poverty at $3.20...	SI.POV.LMIC.GR	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Рис. 4.11. Фрагмент данных из файла

При рассмотрении данных всегда бывает полезно узнать, сколько в них отсутствующих значений и какой процент от всех данных они занимают. В данном случае нас интересуют столбцы начиная с **1974** и далее до конца датафрейма. Метод `isna` возвращает объект `Series` с логическими значениями для каждой колонки. Затем можно получить среднее значение по колонкам, что позволит определить процент отсутствующих значений во всех столбцах с годами, как показано ниже:

```
data.loc[:, '1974':].isna().mean().mean()
0.9184470475910692
```

Как видите, 91,8 % всех ячеек в колонках с годами у нас пустые. Этот факт серьезно влияет на выводы о данных, говоря о том, что в большинстве случаев у нас будет недостаточно сведений по странам, а для некоторых из них данные могут вовсе отсутствовать. Одной из причин может быть то, что многие страны не существовали в их нынешнем виде до начала 1990-х.

Теперь давайте посмотрим, как можно преобразовывать наши датафреймы в длинный формат и обратно и, что более важно, зачем нам это нужно.

Отмена свертывания датафреймов

Наверное, первое, на что вы обратили внимание, – это то, что информация по годам в нашем основном датафрейме хранится в отдельных столбцах – по одному на каждый год. Проблема такого типа хранения данных заключается в том, что `1980`, к примеру, не является полноценной переменной. Гораздо удобнее было бы иметь переменную с именем `year`, в которой были бы собраны все значения показателя за годы с 1974-го по 2019-й. Если помните, мы уже строили график на основе такого типа хранилища, и это было очень удобно. Позвольте мне продемонстрировать то, о чем я говорю, на небольшом примере, а затем мы вместе реализуем этот подход применительно к нашему датафрейму `data`.

На рис. 4.12 показаны два способа хранения одних и тех же данных.

Wide format

	country	indicator	2015	2020
0	country_A	indicator 1	100	120
1	country_B	indicator 1	10	15

Long (tidy) format

	country	indicator	year	value
0	country_A	indicator 1	2015	100
1	country_B	indicator 1	2015	10
2	country_A	indicator 1	2020	120
3	country_B	indicator 1	2020	15

Рис. 4.12. Два датасета с одними и теми же данными в разных форматах

Сейчас наш датафрейм отформатирован в стиле *широкой* (wide) таблицы, а нам хотелось бы преобразовать его в *длинный* (long) формат.

Основная сложность анализа данных в случае с широким форматом хранения информации – в том, что переменные представлены по-разному. Какие-то

из них отображаются вертикально в виде столбца (**country** и **indicator**), тогда как другие содержат данные в горизонтальном виде в двух столбцах: **2015** и **2020**. Осуществлять доступ к одним и тем же данным в случае с длинным форматом хранения очень просто – для этого достаточно указать колонку, в которой они хранятся. Кроме того, в таком формате производится автоматическое сопоставление значений. К примеру, если извлечь из датафрейма столбцы **year** и **value**, значение года 2015 будет автоматически сопоставлено значениям 100 и 10. В то же время каждая строка представлена как полноценное и независимое наблюдение.

К счастью, преобразовать датафрейм к длинному формату можно при помощи одного простого метода `melt`, как показано ниже:

```
wide_df.melt(id_vars=['country', 'indicator'],
             value_vars=['2015', '2020'],
             var_name='year')
```

Посмотрим, каково предназначение каждого из переданных в эту функцию параметров:

- `id_vars`: столбцы, перечисленные в этом параметре, будут сохранены как есть, а значения в них могут быть размножены при необходимости для сохранения соответствий;
- `value_vars`: заголовки перечисленных здесь столбцов будут помещены в одну колонку, а значения – в другую, с полным соблюдением соответствий между ними. Если не указать этот параметр, отмене свертывания будут подвергнуты все столбцы, не перечисленные в переменной `id_vars`;
- `var_name`: необязательный параметр, с помощью которого можно задать имя столбца, в котором будут собраны заголовки исходных колонок.

Теперь давайте применим операцию отмены свертывания к нашему датафрейму `data`:

```
id_vars = ['Country Name', 'Country Code', 'Indicator Name', 'Indicator Code']
data_melt = pd.melt(data, id_vars=id_vars,
                           var_name='year').dropna(subset=['value'])
data_melt['year'] = data_melt['year'].astype(int)
print(data_melt.shape)
data_melt.sample(10)
```

Код получился практически идентичным предыдущему примеру. Сначала мы создали переменную `id_vars`, собрав в списке имена столбцов, которые нужно сохранить. Далее мы передали эту переменную одноименному параметру метода `melt`, после чего удалили отсутствующие значения в столбце `value` при помощи метода `dropna`. Мы могли бы одновременно изменить имя этого столбца, воспользовавшись параметром `value_name`, но имя `value`

вполне нам подходит. В завершение мы привели столбец с годами к числовому типу данных. Запуск этого фрагмента кода приведет к выводу размерности обновленного датафрейма и десяти случайных строк из него, что видно на рис. 4.13.

```
(44417, 6)
```

	Country Name	Country Code	Indicator Name	Indicator Code	year	value
487193	Belgium	BEL	Number of poor at $3.20 a day (2011 PPP) (mill...	SI.POV.LMIC.NO	2015	0.0
218139	Guyana	GUY	Population, total	SP.POP.TOTL	1992	748602.0
431387	Honduras	HND	Population, total	SP.POP.TOTL	2010	8317470.0
246046	Senegal	SEN	Poverty gap at $5.50 a day (2011 PPP) (% of po...	SI.POV.UMIC.GP	1994	62.1
251850	Costa Rica	CRI	Income share held by highest 10%	SI.DST.10TH.10	1995	33.8
430878	Germany	DEU	Poverty gap at $5.50 a day (2011 PPP) (% of po...	SI.POV.UMIC.GP	2010	0.1
518299	Sao Tome and Principe	STP	Population, total	SP.POP.TOTL	2017	207089.0
252830	Ethiopia	ETH	Poverty gap at $5.50 a day (2011 PPP) (% of po...	SI.POV.UMIC.GP	1995	68.9
446409	Pakistan	PAK	Income share held by fourth 20%	SI.DST.04TH.20	2011	21.0
480718	Malta	MLT	Income share held by second 20%	SI.DST.02ND.20	2014	13.5

Рис. 4.13. Датафрейм после отмены свертывания

Первые четыре колонки остались прежними, и все уникальные сочетания значений в них не были затронуты. Что касается столбцов с годами, им повезло меньше, поскольку все они были собраны в две колонки с именами **year** и **value**.

Теперь давайте посмотрим, как можно еще улучшить структуру датафрейма, применив к нужным столбцам обратную операцию.

Сведение датафреймов

Нам было бы гораздо удобнее работать с данными, если бы мы применили к колонке **Indicator Name** операцию, обратную той, что использовали применительно к столбцам с годами. В идеале мы хотели бы видеть отдельно данные по численности населения, уровню бедности и другим показателям. Давайте сначала опробуем эту технику на нашем пробном датафрейме, чтобы вы хорошо поняли, о чем идет речь.

Представьте, что нам необходимо выделить уникальные значения из колонки **country** и создать для каждого из них отдельный столбец. Воспользуемся для этого методом `pivot`. Этот метод дает нам обратный билет туда, откуда мы пришли, применив метод `melt`. Используем его применительно к другим столбцам:

```
melted.pivot(index=['year', 'indicator'],
                columns='country',
                values='value').reset_index()
```

Запуск этого метода позволит преобразовать датафрейм в широкий формат, что видно по рис. 4.14.

Long (tidy) format

	country	indicator	year	value
0	country_A	indicator 1	2015	100
1	country_B	indicator 1	2015	10
2	country_A	indicator 1	2020	120
3	country_B	indicator 1	2020	15

Pivoted (wide) format

country	year	indicator	country_A	country_B
0	2015	indicator 1	100	10
1	2020	indicator 1	120	15

Рис. 4.14. Преобразование датафрейма из длинного формата в широкий

Колонка **Indicator Name** в датафрейме `data_melt` содержит названия индикаторов, которые можно использовать в качестве имен столбцов, чтобы каждый индикатор был представлен отдельно. Этого можно добиться следующим образом:

```
data_pivot = data_melt.pivot(index=['Country Name', 'Country Code', 'year'],
                             columns='Indicator Name',
                             values='value').reset_index()
print(data_pivot.shape)
data_pivot.sample(5)
```

Запуск этого кода приведет к созданию датафрейма `data_pivot`, фрагмент которого показан на рис. 4.15.

```
(8287, 57)
```

Indicator Name	Country Name	Country Code	year	Annualized growth in per capita real survey mean consumption or income, bottom 40% (%)	Annualized growth in per capita real survey mean consumption or income, top 10% (%)	Annualized growth in per capita real survey mean consumption or income, top 60% (%)	Annualized growth in per capita real survey mean consumption or income, total population (%)	Annualized growth in per capita real survey median income or consumption expenditure (%)	GINI index (World Bank estimate)
5872	Paraguay	PRY	2002	NaN	NaN	NaN	NaN	NaN	57.3
853	Brazil	BRA	2017	NaN	NaN	NaN	NaN	NaN	53.3
2665	Germany	DEU	1989	NaN	NaN	NaN	NaN	NaN	NaN
3333	Indonesia	IDN	1983	NaN	NaN	NaN	NaN	NaN	NaN
2747	Greece	GRC	1981	NaN	NaN	NaN	NaN	NaN	NaN

Рис. 4.15. Обновленный вид датафрейма

Если мы все сделали правильно, в каждой строке теперь должна быть представлена уникальная комбинация из страны и года. По сути, в этом и состояла наша конечная цель. Давайте проверим, что мы не ошиблись в ее достижении:

```
data_pivot[['Country Code', 'year']].duplicated().any()
False
```

В строках у нас теперь содержатся названия стран, их коды, годы и значения по каждому индикатору. При этом данные о странах можно обогатить за счет включения метаданных из датафрейма `country`. Давайте взглянем на полезную функцию `merge`, после чего приступим к работе с Plotly Express.

Объединение датафреймов

Сначала посмотрим на упрощенный пример того, как работает объединение датафреймов, после чего применим полученные знания на практике и объединим датафреймы `data_pivot` и `country`. На рис. 4.16 схематично показано, как объединяются датафреймы.

df: "left"

	country	indicator	year	value
0	country_A	indicator 1	2015	100
1	country_B	indicator 1	2015	10
2	country_A	indicator 1	2020	120
3	country_B	indicator 1	2020	15

+

df: "right"

	country	continent	group
0	country_A	Asia	low income
1	country_B	Europe	high income

=

df: merged

	country	indicator	year	value	continent	group
0	country_A	indicator 1	2015	100	Asia	low income
1	country_B	indicator 1	2015	10	Europe	high income
2	country_A	indicator 1	2020	120	Asia	low income
3	country_B	indicator 1	2020	15	Europe	high income

Рис. 4.16. Объединение датафреймов

Объединение выполняется при помощи метода `merge`, как показано ниже:

```
pd.merge(left=left, right=right,
             left_on='country',
             right_on='country',
             how='left')
```

Далее представлена расшифровка параметров метода `pd.merge`:

- `left_on`: имя столбца из левой таблицы, по которому будет производиться объединение датафреймов;
- `right_on`: имя столбца из правой таблицы, по которому будет производиться объединение датафреймов;
- `how`: способ объединения. В нашем случае значение `left` означает, что из левой таблицы при объединении будут взяты все строки, а из правой – только те, в которых есть соответствие левой таблице по ключевому столбцу (**country**). В итоговом датафрейме эти строки могут быть продублированы, как в нашем случае. Строки из правой таблицы, для которых не нашлось совпадений в левой таблице по столбцу **country**, будут отброшены. Результирующий датафрейм будет насчитывать столько же строк, сколько было в левом датафрейме до объединения.

Метод `merge` поддерживает и другие параметры, что делает его довольно мощным. Советуем также ознакомиться с другими методами объединения: `inner`, `outer` и `right`. Итак, давайте объединим датафреймы `data_pivot` и `country` в единый датафрейм `poverty` следующим образом:

```
poverty = pd.merge(data_pivot, country,
                   left_on='Country Code',
                   right_on='Country Code',
                   how='left')
print(poverty.shape)
poverty
```

Выполнение этого кода приведет к образованию датафрейма `poverty`, фрагмент которого показан на рис. 4.17.

(8287, 89)

	Country Name	Country Code	year	Annualized growth in per capita real survey mean consumption or income, bottom 40% (%)	Survey mean consumption or income per capita, top 60% (2011 PPP $ per day)	Survey mean consumption or income per capita, total population (2011 PPP $ per day)	Short Name	Table Name	Long Name	2-alpha code	Currency Unit	Special Notes	Region	Income Group
6251	Sao Tome and Principe	STP	1976	NaN	NaN	NaN	São Tomé and Principe	São Tomé and Principe	Democratic Republic of São Tomé and Principe	ST	São Tomé and Principe dobra	National account data were adjusted to reflect...	Sub-Saharan Africa	Lower middle income
153	Angola	AGO	1992	NaN	NaN	NaN	Angola	Angola	People's Republic of Angola	AO	Angolan kwanza	NaN	Sub-Saharan Africa	Lower middle income
5823	Papua New Guinea	PNG	1998	NaN	NaN	NaN	Papua New Guinea	Papua New Guinea	The Independent State of Papua New Guinea	PG	Papua New Guinea kina	NaN	East Asia & Pacific	Lower middle income
2785	Guatemala	GTM	1974	NaN	NaN	NaN	Guatemala	Guatemala	Republic of Guatemala	GT	Guatemalan quetzal	NaN	Latin America & Caribbean	Upper middle income
175	Angola	AGO	2014	NaN	NaN	NaN	Angola	Angola	People's Republic of Angola	AO	Angolan kwanza	NaN	Sub-Saharan Africa	Lower middle income
4494	Madagascar	MDG	1974	NaN	NaN	NaN	Madagascar	Madagascar	Republic of Madagascar	MG	Malagasy ariary	NaN	Sub-Saharan Africa	Low income

Рис. 4.17. Объединение датафреймов `data_pivot` и `country`

Быстрая проверка того, что мы нигде не ошиблись:

```
poverty[['Country Code', 'year']].duplicated().any()
```
```
False
```

Восемь столбцов, обведенных на рис. 4.17, были добавлены к нашему датафрейму в результате объединения. Число показателей в этом выводе мы искусственно уменьшили, чтобы таблица выглядела более компактно. С данными, представленными в таком формате, очень легко будет работать. Можно взять любой регион или уровень дохода, отфильтровать датафреймы по входящим в них странам, раскрасить значения или сгруппировать их так, как нам понадобится. Теперь наши данные похожи на набор данных `Gapminder`, но с большим количеством индикаторов и лет, а также с ценными метаданными по странам.

Итак, мы получили датафрейм с удобной для анализа структурой.

В каждом столбце теперь хранятся данные об одном и только одном показателе. Все значения в столбцах характеризуются одним типом данных (с на-

личием пропущенных значений). Каждая строка, в свою очередь, представляет одно и только одно наблюдение, в котором присутствуют все необходимые данные.

> **Важно**
>
> Главным недостатком длинного формата данных является его неэффективное хранение. Если смотреть с этой стороны, то в наших данных присутствует избыточное количество повторений, каждое из которых требует отдельного пространства для хранения. Мы еще поработаем над этим позже, а сейчас вам достаточно знать, что в плане анализа данных длинный формат является наиболее эффективным. Как мы уже видели на примерах, после приведения исходных данных к такому формату процесс их анализа и визуализации становится намного более простым и понятным.

Я горячо рекомендую вам ознакомиться со статьей Хэдли Уикема (Hadley Wickham) под названием *Tidy Data* (**https://www.jstatsoft.org/article/view/v059i10**), в которой прекрасно описаны различные форматы хранения данных. Иллюстрируя примеры в этой главе, я основывался на принципах, постулированных в данной работе.

Теперь вы абсолютно готовы к тому, чтобы начать работать с Plotly Express – сначала с тестовым набором данных, а затем и с подготовленным нами ранее.

Знакомство с Plotly Express

Plotly Express представляет собой высокоуровневую графическую систему, созданную поверх Plotly. Она не только позволяет нам не заботиться о подписях для осей и легендах, полагаясь на их значения по умолчанию, но также открывает доступ к использованию наших данных для выражения их атрибутов с использованием визуальных проявлений (размера, цвета, местоположения и т. д.). Это реализуется путем сопоставления атрибутов со столбцами с учетом общей структуры данных. Таким образом, у вас появляется возможность отталкиваться при решении задач от данных, о чем мы говорили в начале главы.

Давайте создадим простой датафрейм:

```
df = pd.DataFrame({
    'numbers': [1, 2, 3, 4, 5, 6, 7, 8],
    'colors': ['blue', 'green', 'orange', 'yellow', 'black', 'gray', 'pink',
'white'],
    'floats': [1.1, 1.2, 1.3, 2.4, 2.1, 5.6, 6.2, 5.3],
    'shapes': ['rectangle', 'circle', 'triangle', 'rectangle', 'circle',
'triangle', 'rectangle', 'circle'],
    'letters': list('AAABBCCC')
})
df
```

Внешний вид этого датафрейма показан на рис. 4.18.

	numbers	colors	floats	shapes	letters
0	1	blue	1.1	rectangle	A
1	2	green	1.2	circle	A
2	3	orange	1.3	triangle	A
3	4	yellow	2.4	rectangle	B
4	5	black	2.1	circle	B
5	6	gray	5.6	triangle	C
6	7	pink	6.2	rectangle	C
7	8	white	5.3	circle	C

Рис. 4.18. Простой пример датафрейма

Обычно диаграммы с помощью Plotly Express создаются посредством вызова типа желаемой диаграммы в качестве функции, например `px.line`, `px.histogram` и т. д. Каждая такая функция принимает на вход определенные параметры в зависимости от типа диаграммы. Существует несколько способов передачи параметров таким функциям, и мы сконцентрируемся на двух главных подходах:

- передача датафрейма с именами столбцов: в большинстве случаев первым передается параметр `data_frame`. Вы передаете датафрейм, который собираетесь визуализировать, а затем указываете столбцы в качестве нужных вам параметров. В нашем случае если мы хотим построить точечную диаграмму с переменной **numbers** на оси X и **floats** на оси Y, мы должны вызвать функцию следующим образом: `px.scatter(data_frame=df, x='numbers', y='floats')`;
- передача массивов: еще одним способом указания параметров является передача на вход функции списков, кортежей или любых массивоподобных структур – без задания параметра `data_frame`. Можно создать такую же диаграмму с помощью следующей записи: `px.scatter(x=df['numbers'], y=df['floats'])`. Это довольно простой и быстрый способ, особенно когда у вас есть в наличии списки, которые нужно проанализировать графически.

Также допустимо сочетать эти подходы. Можно передать параметр `data_frame` с названиями колонок в качестве дополнительных аргументов, а отдельными параметрами указать имеющиеся списки, если это необходимо. Далее мы еще не раз увидим применение данной концепции. С помощью следующего кода можно легко построить точечную диаграмму:

```
px.scatter(df, x='numbers', y='floats')
```

На рис. 4.19 показан выведенный в результате график.

```
1  px.scatter(df, x='numbers', y='floats')
```

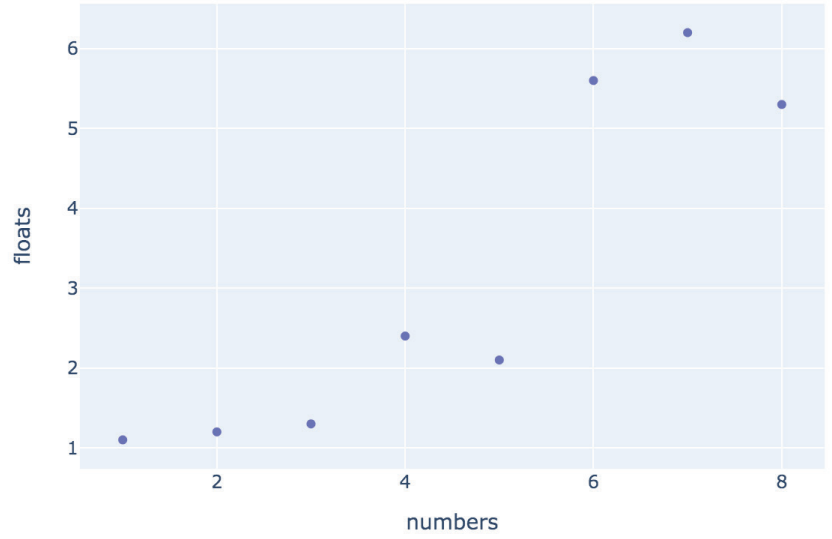


Рис. 4.19. Создание точечной диаграммы с помощью Plotly Express

Думаю, вы обратили внимание, что подписи к осям добавились автоматически. Они были взяты из параметров функции – в нашем случае это имена столбцов в датафрейме.

В нашем датафрейме присутствуют и другие колонки, зависимости которых нам хотелось мы проверить при помощи точечной диаграммы. К примеру, давайте рассмотрим взаимосвязь между переменными **floats** и **shapes**.

Мы можем использовать тот же вызов функции, добавив к нему два параметра для определения того, какой маркер какой *фигуре* (shape) соответствует. Для указания цвета маркеров можно воспользоваться параметром `color`, который поможет раскрасить точки данных в соответствии со значениями в столбце **shapes**. Также можно добавить параметр `symbol`, отвечающий за форму маркеров на диаграмме. Таким образом, мы дадим пользователю сразу два критерия различий для точек данных:

```
px.scatter(df, x='numbers', y='floats',
           color='shapes',
           symbol='shapes')
```

На рис. 4.20 показан вывод этого графика в JupyterLab.

```
px.scatter(df, x='numbers', y='floats',
           color='shapes',
           symbol='shapes')
```

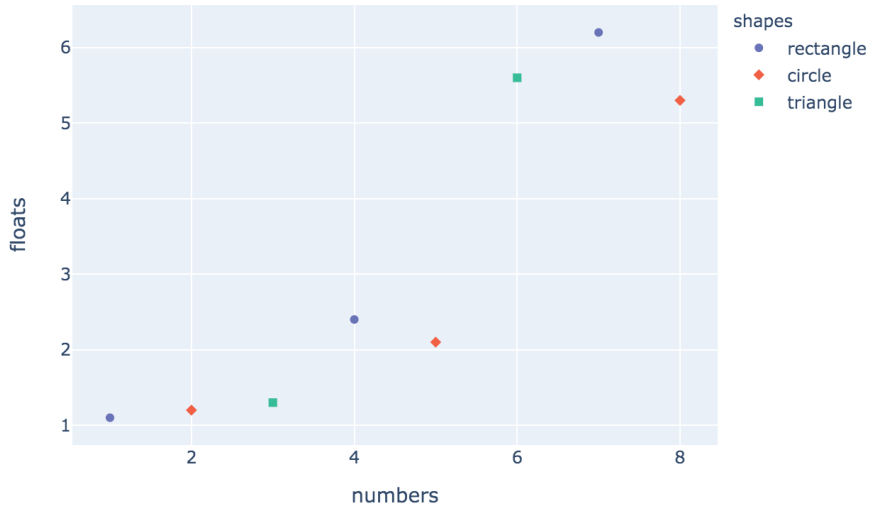


Рис. 4.20. Использование параметров `color` и `symbol` на точечной диаграмме

Теперь на диаграмме также появилась легенда, позволяющая различать точки данных по цвету и форме. Кроме того, у легенды есть и собственный заголовок, созданный автоматически.

Похоже, между переменными **floats** и **shapes** нет никакой четкой зависимости. Давайте попробуем передать переменным `color` и `symbol` значения столбца **letters**:

```
px.scatter(df, x='numbers', y='floats',
           color='letters',
           symbol='letters',
           size=[35] * 8)
```

На рис. 4.21 показан результат запуска этого фрагмента кода.

Как видно на графике, различия в данных обнаружились именно по переменной **letters**. Это демонстрирует легкость, с которой вы можете анализировать свои данные в самых разных разрезах. Заметьте, что на этот раз мы использовали сочетание подходов к заданию параметров функции, передав в качестве аргумента `size` отдельный список. Параметр `size` не привязан ни к одной из переменных в нашем датафрейме, а служит лишь для увеличения размера точек данных на графике. Таким образом, ему мы просто передали список из восьми чисел 35. Длина этого списка должна совпадать с количеством элементов в переменных, которые мы визуализируем.

```
px.scatter(df, x='numbers', y='floats',
           color='letters',
           symbol='letters',
           size=[35] * 8)
```

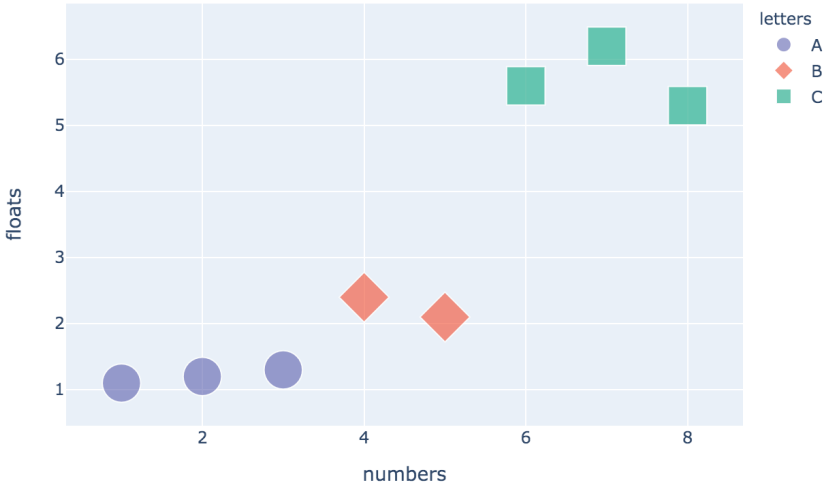


Рис. 4.21. Размер маркеров задан независимым списком

Давайте теперь построим *столбчатую диаграмму* на основе нашего датафрейма, используя тот же подход. Тип отображения диаграммы можно задать при помощи параметра `barmode`, как показано ниже:

```
px.bar(df, x='letters', y='floats', color='shapes', barmode='group')
```

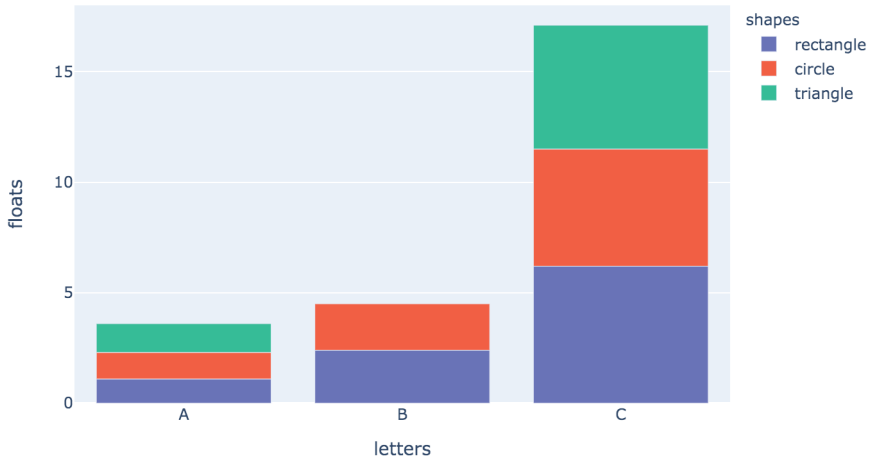
На рис. 4.22 показаны два способа отображения столбчатой диаграммы: в виде накопительных столбиков (по умолчанию) и с группировкой (`barmode='group'`).

Наши рассуждения о длинном формате датафреймов должны поспособствовать лучшему пониманию принципов работы Plotly Express. Все, что вам нужно, – это понимать, как отображается тот или иной тип диаграммы, и использовать нужные параметры при вызове функций.

Важно

Plotly Express не требует, чтобы данные обязательно были приведены к длинному формату. Это очень гибкий инструмент, способный корректно обрабатывать длинный, широкий и смешанный формат. Неменьшей гибкостью в области преобразования данных обладают пакеты `pandas` и `numpy`. Но более эффективно будет придерживаться какого-то одного подхода при форматировании исходных данных.

```
1 px.bar(df, x='letters', y='floats', color='shapes')
```




```
1 px.bar(df, x='letters', y='floats', color='shapes', barmode='group')
```

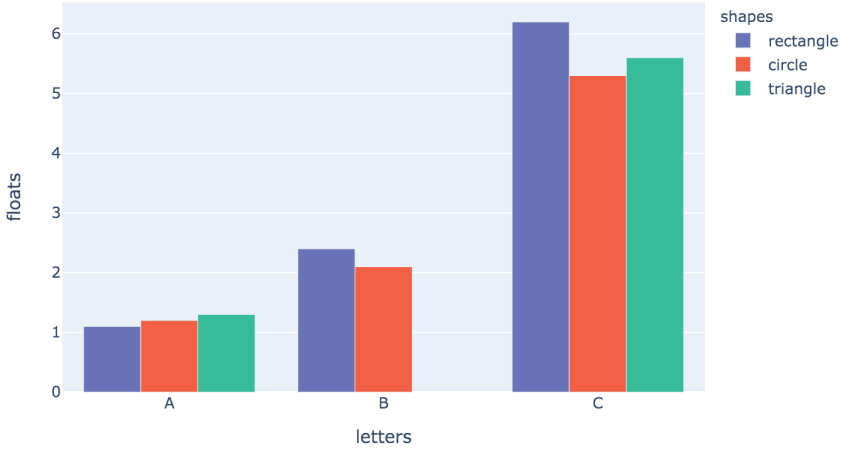


Рис. 4.22. Два режима отображения столбиков на столбчатой диаграмме

Теперь посмотрим, как Plotly Express взаимодействует с объектом `Figure` и когда стоит применять их совместно.

Plotly Express и объекты Figure

Важно понимать, что все вызовы функций построения графиков Plotly Express возвращают объект `Figure` – тот самый, который мы обсуждали в гла-

ве 3. Это открывает дополнительные возможности для настройки диаграмм уже после их фактического создания, например если вы хотите изменить какие-то значения по умолчанию. К примеру, вы можете построить точечную диаграмму, после чего добавить на нее определенные аннотации. Это можно сделать так, как мы уже делали в предыдущей главе:

```
import plotly express as px
fig = px.scatter(x=[1, 2, 3], y=[23, 12, 34])
fig.add_annotation(x=1, y=23, text='This is the first value')
fig
```

Все, что вы уже знаете об объекте `Figure`, может быть применено и в работе с Plotly Express.

Но тут возникает резонный вопрос: когда необходимо использовать Plotly Express, а когда – модуль `graph_objects` из Plotly при построении диаграмм? Этот вопрос можно обобщить и переформулировать так: как выбрать из двух интерфейсов, выполняющих одни и те же действия на разных уровнях абстракции?

Рассмотрим следующие три способа съесть пиццу:

- **доставка**: вы просто звоните в пиццерию, и вам привозят готовую пиццу. В течение получаса пицца у вас на пороге, и вы сразу можете ее есть;
- **супермаркет**: вы идете в супермаркет, покупаете тесто, сыр, овощи и остальные ингредиенты, приходите домой и делаете пиццу самостоятельно;
- **ферма**: вы выращиваете томаты на заднем дворе, заводите коров, доите их, превращаете молоко в сыр и т. д.

При повышении уровня интерфейса до доставки вам требуется все меньше знаний и умений. За качество пиццы отвечает ресторан, действуя в рамках здоровой конкуренции. Недостатком такого подхода является весьма высокая ограниченность выбора. У каждого ресторана есть определенный ассортимент из видов пиццы, и ничего другого вы выбрать просто не можете.

При движении по иерархии интерфейсов ниже требования к уровню осведомленности постепенно повышаются, увеличивается степень ответственности за результат, достижение которого требует больше времени. Взамен же мы получаем больше свободы выбора и меньшую цену итогового продукта, если он производится в достаточно большом объеме. Если вы просто хотите полакомиться пиццей, вам будет дешевле заказать ее в ресторане. Но если вы планируете есть пиццу ежедневно, самостоятельное ее изготовление может существенно сэкономить вам средства.

Такого рода компромисс лежит и в основе выбора между Plotly Express и низкоуровневым модулем Plotly `graph_objects`.

Поскольку функции Plotly Express возвращают объект `Figure`, обычно ответ на поставленный вопрос не требует много времени, ведь возвращаемые объекты можно модифицировать уже после их создания. Обычно рекомендуется прибегать к помощи модуля `graph_objects` в следующих случаях:

- **нестандартная визуализация**: многие диаграммы из этой книги были построены при помощи Plotly. Создать их посредством Plotly Express было бы довольно затруднительно в силу их нестандартности;
- **необходимость в тонких настройках**: если после создания диаграммы вам нужно внести множество изменений, вы все равно потратите на это достаточно много времени и усилий, так что в этом случае вы можете использовать модуль `graph_objects` напрямую;
- **наличие множества графиков**: иногда вам необходимо размещать несколько графиков бок о бок или в табличном виде. Если речь идет об отдельных графиках, а не о составных, как в начале этой главы, то лучше сразу воспользоваться модулем `graph_objects`.

В целом библиотека Plotly Express обычно является неплохой отправной точкой для входа в мир построения графиков. Вы уже видели, как просто ей пользоваться и какой мощью она обладает.

Итак, вы уже готовы, чтобы начать визуализировать данные из датафрейма `poverty` при помощи модуля Plotly Express.

Создание диаграммы Plotly Express на основе набора данных

Давайте попробуем визуализировать датафрейм `poverty` с помощью нашей любимой точечной диаграммы.

1. Создайте переменные `year`, `indicator` и `grouper` для соответствующих показателей – их мы будем использовать в нашей визуализации. Группирующая метрика будет применяться для внесения различий в отображение точек данных (с помощью параметров `color` и `symbol`) и может принимать любое категориальное значение из датасета, будь то регион, уровень дохода и пр.:

```
year = 2010
indicator = 'Population, total'
grouper = 'Region'
```

2. На основе этих переменных создайте датафрейм, в котором должна остаться только информация об указанном годе, а пропущенные значения в полях `indicator` и `grouper` должны быть исключены:

```
df = (poverty[poverty['year'].eq(year)]
        .sort_values(indicator)
        .dropna(subset=[indicator, grouper]))
```

3. Передайте функции `px.scatter` в качестве параметра `x` переменную `indicator`, а параметра `y` – название столбца `"Country Name"`. Тем самым мы определяем, что у нас будет находиться на осях. Параметрам `color` и `symbol` мы присвоим переменную `grouper`. Мы ожидаем, что значения

на оси *x* не будут распределены нормально и будут содержать выбросы, так что сделаем ось логарифмической, присвоив параметру `log_x` значение `True`. В качестве параметра `hover_name` передадим поля **Short Name** и **flag**. В заголовке графика (параметр `title`) объединим через пробел `indicator`, `"by"`, `grouper` и `year` при помощи метода `join`. В качестве параметра `size` передадим список единиц нужного размера, а высоту диаграммы (параметр `height`) ограничим 700 пикселями. Все это показано во фрагменте кода ниже:

```
px.scatter(data_frame=df,
           x=indicator,
           y='Country Name',
           color=grouper,
           symbol=grouper,
           log_x=True,
           hover_name=df['Short Name'] + ' ' + df['flag'],
           size=[1]* len(df),
           title= ' '.join([indicator, 'by', grouper, str(year)]),
           height=700)
```

В результате вы получите график, показанный на рис. 4.23.

Рис. 4.23. График, построенный при помощи Plotly Express

Пробуя разные комбинации переменных `year`, `grouper` и `indicator`, вы можете получать сотни самых разных графиков. На рис. 4.24 показаны лишь некоторые из них.

Если ваши данные аккуратно форматированы так, что в каждой строке хранится одно наблюдение, а в каждом столбце – переменная, вы можете легко визуализировать шесть или семь подобных переменных с использованием таких визуальных атрибутов, как оси *x* и *y*, размер маркеров, используемый символ, разделение на подграфики (по строкам или колонкам) и анимация.

Дополнительный контекст можно задать при помощи всплывающих окон при наведении мышью и аннотаций. Любую комбинацию этих переменных можно визуализировать отдельно путем выбора соответствий между колонками датафрейма и атрибутами.

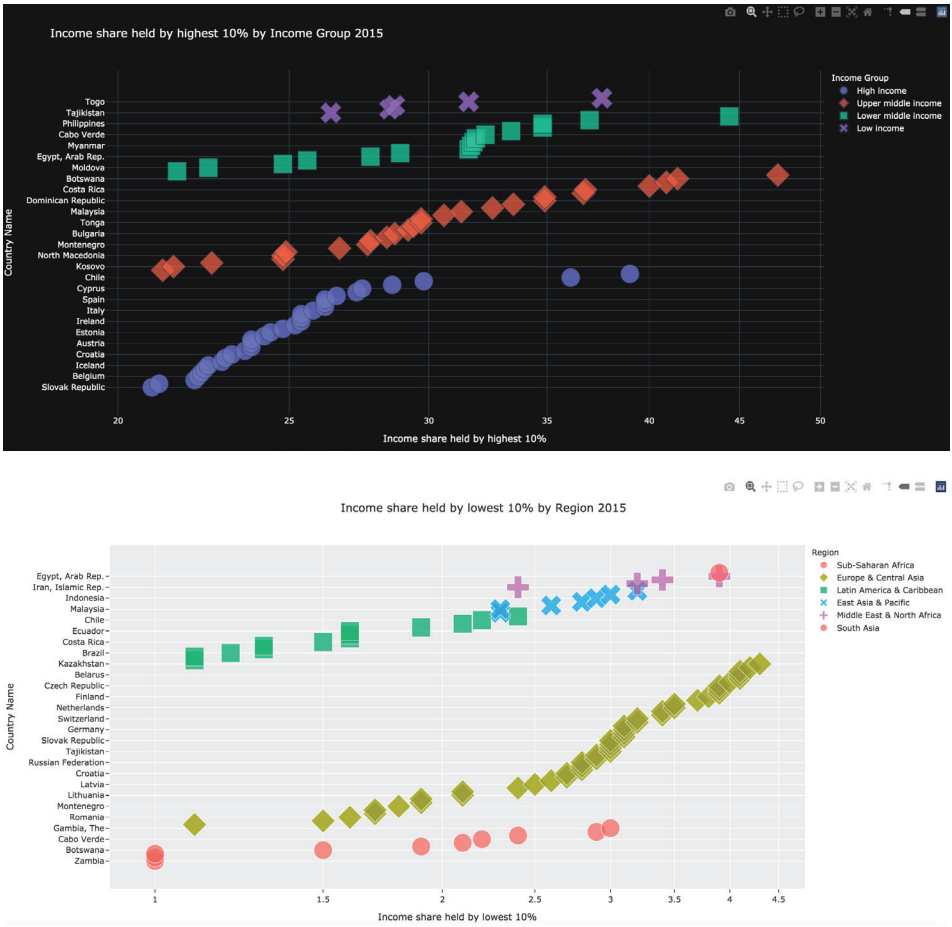


Рис. 4.24. Графики, построенные с использованием того же датафрейма

Теперь давайте узнаем, как можно обогатить набор данных с помощью внешних ресурсов.

Добавление данных и столбцов в набор

Существует масса способов добавить новые данные в набор, но мы остановимся на двух наиболее простых и эффективных:

- **добавление столбцов или переменных**: мы с вами уже добавили в наш датафрейм колонку с флагами, используя для этого двухсимвольные обозначения стран. Вы также можете осуществить сегментирование или группировку стран по определенным атрибутам. Возможно, вы

ожидаете увидеть какие-то важные отличия в странах, использующих в качестве валюты евро или говорящих на испанском либо арабском языке. Или вас интересуют страны, подписавшие какое-то торговое соглашение. В Википедии содержится огромное множество разнообразных наборов данных. Используя функцию `read_html` из пакета `pandas`, позволяющую загружать табличные данные из интернета, вы можете без труда скачать любой нужный вам список. А если в этом списке будет присутствовать код страны, вы спокойно объедините загруженные данные с нашим основным датафреймом и проанализируете их в совокупности. Эти дополнительные сведения можно также использовать в качестве фильтров, чтобы оставить в итоговом наборе ограниченное количество стран;

- **добавление новых данных**: на сайте Всемирного банка (World Bank) также присутствует немало полезных наборов данных. К примеру, тот набор, который мы анализировали в этой главе, содержит информацию о численности населения в странах в целом. На сайте же можно загрузить эту информацию в разрезах по полу, возрасту и другим факторам. Используя API сайта Всемирного банка, вы с легкостью можете получить все интересующие вас сведения и объединить их с существующими наборами данных, тем самым значительно обогатив их аналитический потенциал.

Давайте резюмируем, что мы узнали из этой главы и всей первой части книги.

Заключение

На данный момент вы обладаете достаточной информацией и рассмотрели довольно много примеров быстрого создания элементов дашборда. В главе 1 мы познакомились с общей структурой приложений Dash и узнали, что нужно сделать для создания и запуска полноценного приложения, не обладающего интерактивностью. В главе 2 мы говорили о том, как осуществляется взаимодействие между пользователем и приложением, познакомились с функциями обратного вызова и добавили написанному нами ранее приложению интерактивности. Третья глава была целиком посвящена тому, как именно создаются графики Plotly, из каких компонентов они состоят и как ими управляют для достижения нужных целей. Наконец, в главе 4 мы познакомились с верхнеуровневым интерфейсом Plotly под названием Plotly Express, позволяющим очень легко создавать довольно сложные визуализации. При этом главным преимуществом этого инструмента является его ориентированность на данные, а не на диаграммы.

Одним из важнейших процессов на пути создания визуализаций является подготовка исходных данных и преобразование их в формат, делающий дальнейший анализ простым и понятным. Отведение достаточного количества времени на понимание структуры ваших исходных данных и их трансформацию в нужный вам формат впоследствии сэкономит вам массу времени и сил в процессе их визуализации, как мы видели в этой главе.

Вооружившись полученными знаниями и навыками, а также нашим набором данных, который уже должен быть вам хорошо знаком, вы можете приступать к углубленному изучению различных компонентов Dash, а вместе с ними и новых типов диаграмм.

Вторая часть книги будет посвящена работе с разными типами графиков и способам их комбинирования с интерактивными возможностями, предоставляемыми компонентами Dash.

Часть II

Расширение функционала приложений

В этой части вы начнете работать с реальными данными и научитесь применять богатый функционал Dash в области интерактивности.

Содержание этой части:

- глава 5 *«Интерактивное сравнение данных при помощи столбчатых диаграмм и выпадающих списков»;*
- глава 6 *«Исследование переменных при помощи точечной диаграммы и фильтрация наборов данных»;*
- глава 7 *«Работа с географическими картами и обогащение дашбордов при помощи языка разметки Markdown»;*
- глава 8 *«Определение частотности данных с помощью гистограмм и построение интерактивных таблиц».*

Глава 5

Интерактивное сравнение данных при помощи столбчатых диаграмм и выпадающих списков

Итак, у вас на данный момент есть все необходимое для конструирования динамически связанных элементов и создания интерактивных дашбордов. Основные концепции были продемонстрированы с помощью нескольких примеров, а сейчас пришло время познакомиться с другими видами диаграмм и узнать, какие возможности они предлагают. Кроме того, мы погрузимся в изучение дополнительных опций по кастомизации наших графиков и сделаем возможной их публикацию и свободное распространение в противовес локальному использованию, а также оптимальное размещение на экране в соседстве с другими компонентами. Отдельное внимание мы уделим вопросам, связанным с интерактивным потенциалом создаваемых вами графиков. Основой для наполнения визуализаций должен служить исключительно выбор пользователя, а количество отображаемых переменных может быть как 7, так и 70. Бывают и случаи, когда данных для отображения при определенном выборе пользователя просто нет. Это может затруднять чтение визуализации и снижать ее эффективность. Мы рассмотрим несколько вариантов разрешения данной ситуации.

Иными словами, мы постепенно будем продвигаться от работы с прототипами, призванными делать то, для чего созданы, к созданию реального продукта, который можно будет опубликовать для широкой аудитории.

В главах второй части книги мы будем подробно обсуждать различия между разными типами диаграмм и исследовать их возможности. В данной главе мы детально рассмотрим столбчатые диаграммы и их интерактивные возможности на примере взаимодействия с выпадающими списками из состава **Dash Core Components**. При этом в природе этих компонентов нет ничего такого, что связывало бы их с конкретными типами диаграмм. Они просто могут быть использованы совместно в рамках организационной структуры. Выпадающие списки могут взаимодействовать с любыми типами диаграмм, а столбчатая

диаграмма открыта для интерактивного влияния со стороны любого компонента.

Темы, которые будут рассмотрены в главе:

- построение вертикальных и горизонтальных столбчатых диаграмм;
- связывание столбчатых диаграмм с выпадающими списками;
- разные способы отображения столбчатых диаграмм с несколькими рядами данных;
- использование ячеистой структуры для вывода множественных диаграмм;
- исследование дополнительных возможностей выпадающих списков (множественный выбор, заместители текста и т. д.).

Технические требования

В данной главе мы продолжим использовать пакеты JupyterDash и Dash, с которыми начали работать ранее, для создания прототипов решений с их последующей интеграцией в наше приложение. Для преобразования данных мы продолжим использовать пакет pandas, а библиотека JupyterLab будет испытательным аэродромом для наших решений. Также мы воспользуемся пакетами Dash Core Component, Dash HTML Components и Dash Bootstrap Components для обновления нашего приложения.

В работе мы будем пользоваться датафреймом `poverty`, который создали в предыдущей главе.

Исходный код к этой главе располагается в хранилище GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_05**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3ebv8sk**.

Давайте начнем с описания двух способов отображения столбчатых диаграмм: вертикального и горизонтального.

Построение вертикальных и горизонтальных столбчатых диаграмм

По умолчанию столбчатые диаграммы отображаются вертикально. Это наиболее привычный и интуитивно понятный вид такого типа диаграммы. Каждая категория или элемент занимает отдельную позицию на оси *x*, а высота столбиков с соответствующими отметками на оси *y* характеризует количественную характеристику анализируемого параметра в разрезе конкретного значения. Тот же принцип сохраняется и при горизонтальном размещении диаграммы, но в этом случае значения переменных характеризуются не высотой столбиков, а их длиной по горизонтали. Обычно вертикальная столбчатая диаграмма предпочтительнее смотрится при наличии небольшого количества номинативных переменных. В то же время горизонтальная ориентация графика может оказаться более выгодной в следующих случаях:

- **когда в анализ входит множество категорий**: в этом случае столбики могут просто не поместиться на экране, в результате чего может понадобиться сделать их более узкими, что лучше подходит для горизонтальной ориентации. Кроме того, в таких условиях для графика может быть отображена полоса прокрутки, которая, опять же, более естественно смотрится на горизонтальной диаграмме;
- **когда имена категорий относительно длинные**: это не самая большая проблема, решить которую не так трудно. Plotly старается заботиться о пользователях и автоматически меняет угол расположения подписей к категориям при необходимости. Для максимальной оптимизации пространства подписи могут быть расположены даже вертикально. Однако читать перевернутый текст бывает не слишком комфортно, так что в этих случаях зачастую лучше будут подходить горизонтальные столбики, позволяющие сохранить ориентацию подписей.

Давайте рассмотрим процесс создания столбчатой диаграммы на практике с использованием нашего датафрейма `poverty`, чтобы увидеть все своими глазами и лучше понять содержание нашего набора данных. Мы будем анализировать один из наиболее популярных показателей степени развития стран – *индекс Джини*, отражающий степень неравенства в распределении доходов внутри государства. Также эту метрику иногда называют коэффициентом Джини, или статистическим показателем степени расслоения общества. Мы поработаем с датафреймом `series`, содержащим информацию о нужных нам индикаторах.

1. Импортируйте библиотеку `pandas` и создайте переменную `series`. Имя переменной было выбрано на основе имени исходного файла, как мы условились в предыдущей главе. Пожалуйста, не путайте эту переменную с объектом `pandas.Series`:

```
import pandas as pd
series = pd.read_csv('data/PovStatsSeries.csv')
```

2. Создайте переменную `gini`, которая позволит в сокращенном виде использовать название длинного показателя:

```
gini = 'GINI index (World Bank estimate)'
```

3. Извлеките длинное определение индикатора из столбца с именем `Long definition`:

```
series[series['Indicator Name']==gini]['Long definition'].values[0]
```

   Это поможет больше узнать об индексе Джини, который мы собираемся анализировать.

4. Из описания индекса мы узнали, что его значения могут варьироваться в интервале от 0 до 100, а минимальное и максимальное значения узнать довольно просто, выполнив следующий код (здесь мы обращаемся к датафрейму `poverty`, созданному в предыдущей главе путем объединения источников данных):

```
poverty[gini].min(), poverty[gini].max()
(20.2, 65.8)
```

5. Также мы можем узнать немного больше о содержимом интересующего нас столбца, воспользовавшись методом `describe`, как показано ниже:

```
poverty[gini].describe()
count    1674.000000
mean       38.557766
std         9.384352
min        20.200000
25%        31.300000
50%        36.400000
75%        45.275000
max        65.800000
Name: GINI index (World Bank estimate), dtype: float64
```

В этой главе мы будем анализировать индекс Джини и сравнивать по нему различные страны, но стоит при этом иметь в виду важное сообщение, содержащееся в столбце `Limitations and exceptions`:

> *«В связи с разницей применяемых методов и способов сбора информации о благосостоянии населения в разных странах не стоит применять этот показатель для сравнения и объективного суждения о различиях в уровне бедности в этих государствах, а также в рамках одной страны в разные годы».*

Что ж, будем осторожны в своих суждениях при анализе с учетом этого предостережения.

Теперь, когда мы достаточно хорошо знакомы с нашим индикатором, рассмотрим различные опции в процессе его визуализации при помощи столбчатой диаграммы.

1. Начнем с создания поднабора данных на основе датафрейма `poverty`, который назовем `df`. В нем мы соберем данные только по одному произвольно выбранному году. Также мы избавимся от пропущенных значений и отсортируем датафрейм по столбцу с индикатором:

```
year = 1980
df = poverty[poverty['year']==year].sort_values(gini).
dropna(subset=[gini])
```

2. Теперь мы можем создать столбчатую диаграмму для анализа индекса Джини при помощи модуля Plotly Express. Попутно динамически соберем заголовок для визуализации:

```
import plotly.express as px
px.bar(df,
       x='Country Name',
```

```
            y=gini,
            title=' - '.join([gini, str(year)]))
```

Запуск этого фрагмента кода приведет к отображению столбчатой диаграммы, показанной на рис. 5.1.

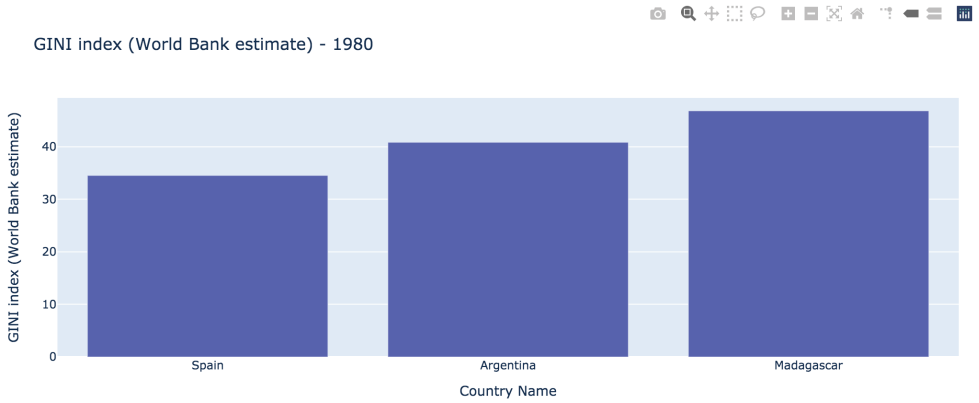


Рис. 5.1. График с индексом Джини за 1980 год

Похоже, в 1980 году данные в датафрейме были только по трем странам: Испании, Аргентине и Мадагаскару. И их отображение на вертикальной столбчатой диаграмме смотрится довольно гармонично. А теперь давайте посмотрим на картину 1990 года, отраженную на рис. 5.2.

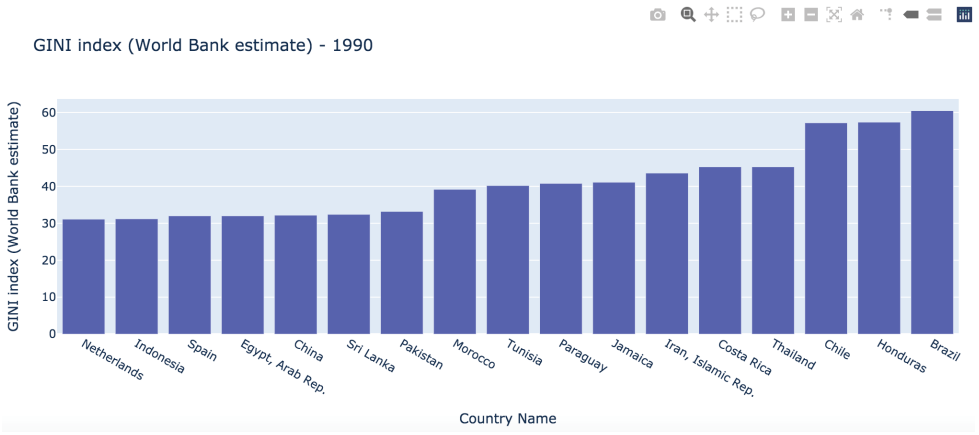


Рис. 5.2. График с индексом Джини за 1990 год

Названия стран, наклоненные на 45 градусов, все еще можно прочитать, но уже не так комфортно, как на рис. 5.1. А если открыть тот же график на более узком экране, подписи и вовсе будут расположены вертикально, как показано на рис. 5.3, что еще больше затруднит их чтение.

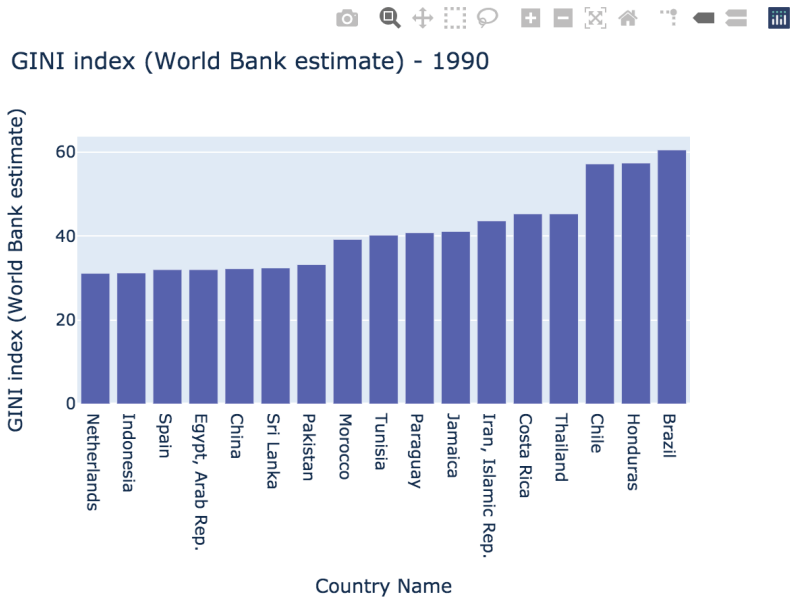


Рис. 5.3. График с индексом Джини за 1990 год с вертикальными подписями

За следующие годы источник данных пополнился сведениями о еще большем количестве стран, и их стало столько, что для них просто перестало хватать места по горизонтали. В результате названия некоторых стран просто пропали, и они не появятся, пока не наведешь на соответствующие столбики мышью или не увеличишь фрагмент диаграммы. В качестве примера посмотрите на рис. 5.4, на котором изображена диаграмма за 2010 год.

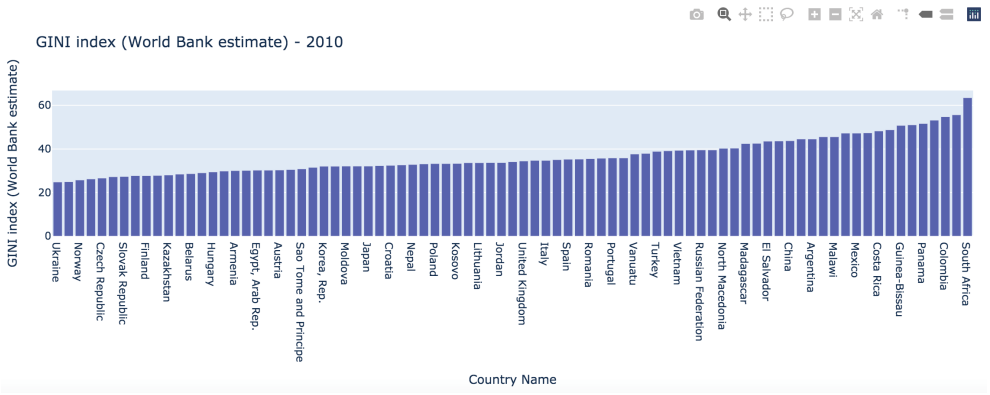


Рис. 5.4. График с индексом Джини за 2010 год – названия некоторых стран отсутствуют

Внимательно рассмотрев все эти примеры, можно сделать вывод о том, что на пути добавления диаграмме с индексом Джини интерактивности есть определенные сложности. Если мы хотим, чтобы пользователь сам выбирал год для анализа, придется немного поработать над внешним видом графика.

Во-первых, количество анализируемых элементов в данном случае может варьироваться в довольно широком диапазоне – от 3 до 150. Во-вторых, было бы гораздо удобнее и более надежно использовать горизонтальную ориентацию диаграммы, поскольку в этом случае названия стран всегда будут легко читаться, какими бы длинными они ни были. Это можно очень просто реализовать, передав параметр `orientation='h'` функции `px.bar`. Но одна проблема все же остается, и заключается она в необходимости определить оптимальную высоту диаграммы на основании количества стран в конкретном выбранном году и с учетом того, как велик диапазон возможных значений. Давайте сначала посмотрим, как будет выглядеть наша визуализация в горизонтальном виде, после чего постараемся добавить ей интерактивности. Внесем в наш код два незначительных изменения: параметры `x` и `y` поменяем местами, а также зададим явное значение параметру `orientation`, как показано ниже:

```
year = 2000
df = poverty[poverty['year']==year].sort_values(gini).dropna(subset=[gini])
px.bar(df,
        x=gini,
        y='Country Name',
        title=' - '.join([gini, str(year)]),
        orientation='h')
```

В результате будет выведен график, показанный на рис. 5.5.

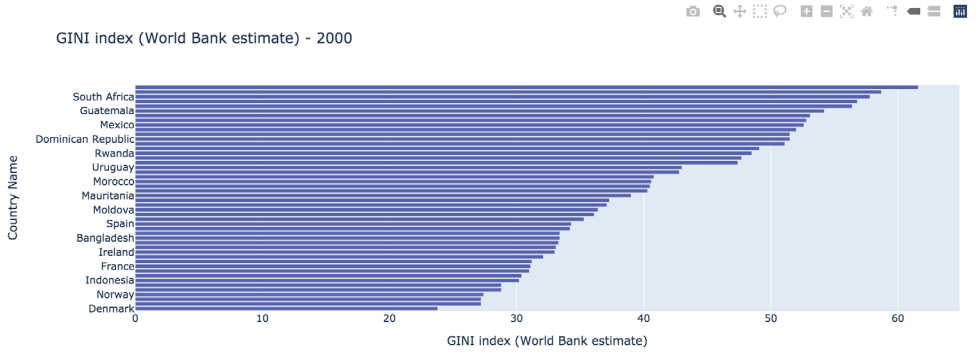


Рис. 5.5. Горизонтальный график с индексом Джини за 2000 год

Названия стран читаются нормально (по крайней мере те, которые выводятся), но сами столбики довольно узкие, из-за чего получается тесновато. К тому же график выглядит несколько растянутым по горизонтали, особенно со знанием того, что действительные значения на нем укладываются в интервал [20.2, 65.8]. Можно изменить ширину графика вручную с помощью специального параметра, тогда как для высоты нужно придумать какой-то динамический способ задавать высоту в зависимости от количества выводимых стран.

Самый простой способ – задать фиксированную высоту в пикселях. Затем, основываясь на количестве строк в датафрейме, добавить по 20 пикселей на каждую страну. К примеру, если у нас 10 стран в датафрейме `df`, высота будет 200 + (10*20) = 400 пикселей. После создания датафрейма мы сразу можем подсчитать количество строк в нем и сохранить его в переменной `n_countries`. Измененный код будет выглядеть так:

```
year = 2000
df = poverty[poverty['year']==year].sort_values(gini).dropna(subset=[gini])
n_countries = len(df['Country Name'])
px.bar(df,
       x=gini,
       y='Country Name',
       title=' - '.join([gini, str(year)]),
       height=200 + (20*n_countries),
       orientation='h')
```

Запуск этого кода с тремя различными годами с разным количеством стран привел к выводу графиков, показанных на рис. 5.6.

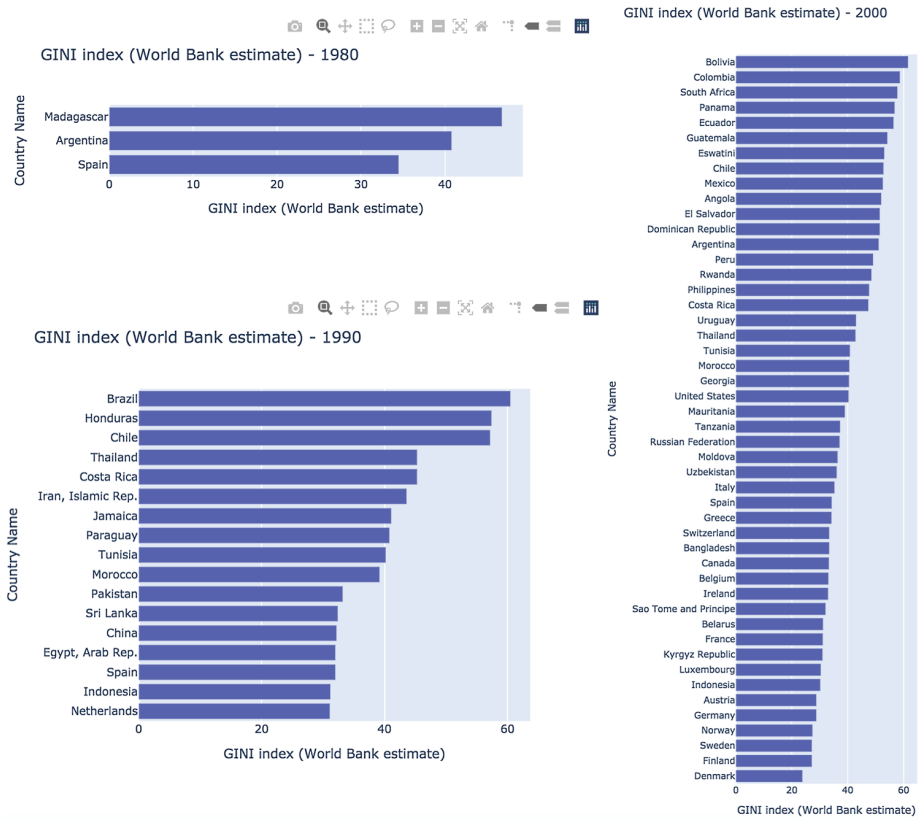


Рис. 5.6. Горизонтальные столбчатые диаграммы с динамической высотой

Длинная диаграмма справа была сжата, чтобы уместиться на страницу, но в плане высоты столбиков и читаемости названия стран она такая же, как и две другие. Все страны видны, их названия хорошо различимы, и ни одна страна не оказалась скрыта.

Этот подход основывается на динамической установке высоты графика в зависимости от количества анализируемых элементов и напрямую связан с исследованием. Пользователь не знает заранее, что именно он будет искать. Он просто выбирает год и смотрит, какая информация будет ему доступна. После этого он может заинтересоваться более подробной информацией о той или иной стране. К примеру, ему может понадобиться узнать, как индекс Джини менялся в конкретной стране с течением времени. Позже мы позволим ему сделать это.

Создание вертикальных столбчатых диаграмм со множеством значений

Если вам нужно продемонстрировать пользователю динамику изменения индекса Джини (или любого другого индикатора) с течением времени, вы можете сделать это при помощи вертикальной столбчатой диаграммы. Поскольку годы представляют собой последовательность временных отметок, отображение их на оси *x* можно назвать вполне естественным, ведь оно хорошо демонстрирует хронологию событий. К тому же годы представлены всего четырьмя цифрами, что позволяет избежать сложностей с отображением подписей. Даже если столбики будут чрезмерно узкими и не все значения по оси *x* смогут быть выведены, пользователь мысленно очень легко заполнит недостающие годы и достроит последовательность.

Код для построения такой диаграммы будет очень похож на предыдущий, но он будет проще, поскольку нам не нужно будет заботиться о динамическом задании высоты графика. К тому же в качестве динамической переменной у нас здесь будет не год, а имя страны, представленное столбцом `Country Name`. Содержание датафрейма будет зависеть от строк, входящих в набор данных по конкретной стране:

```
country = "Sweden"
df = poverty[poverty['Country Name']==country].dropna(subset=[gini])
```

Теперь можно легко и просто построить график с помощью приведенного ниже вызова функции:

```
px.bar(df,
        x='year',
        y=gini,
        title=' - '.join([gini, country]))
```

Результат запуска показан на рис. 5.7.

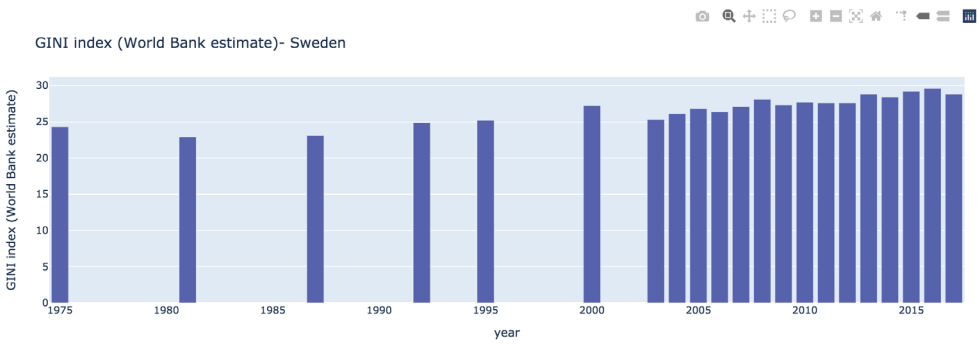


Рис. 5.7. Вертикальная столбчатая диаграмма с годами на оси *x*

Эту диаграмму можно очень легко преобразовать в линейный график – для этого достаточно заменить функцию `px.bar` на `px.line`. Все параметры при этом могут оставаться неизменными.

Заметьте, что годы с отсутствующими значениями в датафрейме все равно представлены на оси *x*, несмотря на отсутствие самих столбиков. Это очень важно, поскольку помогает понять, где на временной оси были пропуски. Если бы мы отображали только те годы, по которым в датафрейме присутствуют фактические значения, это могло бы сбить пользователя с толку, давая понять, что в значениях есть строгое постоянство.

Итак, в этом разделе вы познакомились с индексом Джини на примере двух типов столбчатых диаграмм. Теперь мы готовы добавить в наше приложение соответствующую секцию, чем сейчас и займемся.

Связывание столбчатых диаграмм с выпадающими списками

В этом разделе мы постараемся связать воедино все, что сделали до этого. Давайте разместим бок о бок два выпадающих списка, а под ними соответствующие диаграммы. В левом списке пользователь сможет выбрать год, после чего будет создана горизонтальная столбчатая диаграмма с индексами Джини по странам за этот год. В правом выпадающем списке можно будет выбрать страну и посмотреть динамику индекса Джини в ней по годам. В результате мы получим целую секцию в нашем приложении, показанную на рис. 5.8.

Давайте создадим полноценное приложение в JupyterLab и убедимся, что оно исправно работает.

1. Для начала, как и всегда, импортируем все необходимые пакеты и создадим экземпляр приложения. Мы уже описывали все использованные здесь пакеты, разве что за исключением `dash.exceptions`, из которого мы заберем функцию `PreventUpdate`. Это очень полезный инструмент, позволяющий избежать проблем в случае отсутствующего выбора в компоненте, связанном с функцией обратного вызова. К примеру, это возможно при первой загрузке приложения и отсутствующих значениях полей по умолчанию. В этом случае входящее значение из компо-

нента Dropdown будет None, что, скорее всего, приведет к возникновению исключительной ситуации. Но мы можем использовать возникшее исключение для своеобразной заморозки до момента ввода корректного входного значения:

```
from jupyter_dash import JupyterDash
import dash_html_components as html
import dash_core_components as dcc
import dash_bootstrap_components as dbc
from dash.dependencies import Output, Input
from dash.exceptions import PreventUpdate

app = JupyterDash(__name__)
```

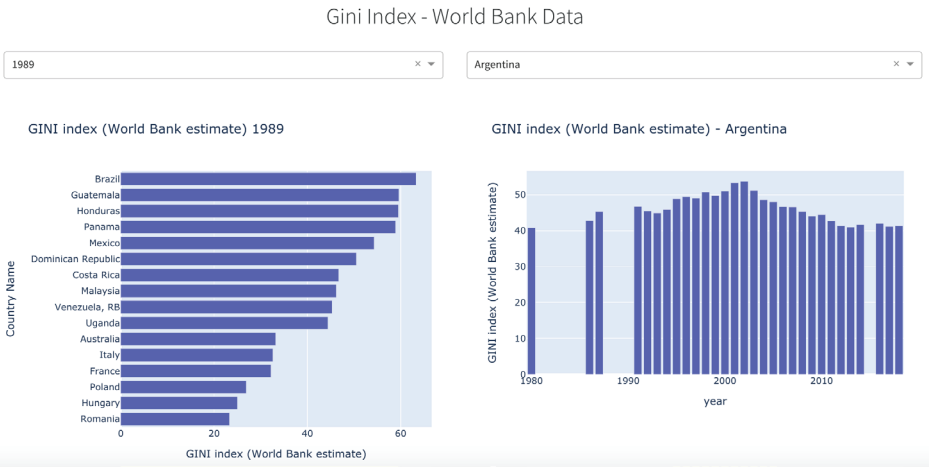


Рис. 5.8. Секция с графиками по индексу Джини с двумя списками

2. Теперь создадим датафрейм с именем gini_df, в котором оставим все строки из датафрейма poverty, за исключением пропущенных значений в столбце с индексом:

```
gini_df = poverty[poverty[gini].notna()]
```

3. Далее создадим макет приложения с использованием одного верхнеуровневого элемента Div, в котором разместим все наши компоненты:

```
app.layout = html.Div()
```

4. В элемент Div добавим заголовок секции и компонент dbc.Row. Внутри него будут находиться компоненты dbc.Col, в которых и будут располагаться наши выпадающие списки и диаграммы. Ниже показан список элементов, который необходимо поместить в верхнеуровневый Div:

```
app.layout = html.Div([
    [
```

```
      html.H2('Gini Index - World Bank Data',
               style={'textAlign': 'center'}),
      dbc.Row([
          dbc.Col([
              dcc.Dropdown(id='gini_year_dropdown',
                           options=[{'label': year, 'value': year}
                           for year in gini_df['year'].
drop_duplicates().sort_values()]),
                  dcc.Graph(id='gini_year_barchart')]),
          dbc.Col([
              dcc.Dropdown(id='gini_country_dropdown',
                           options=[{'label': country, 'value': country}
                           for country in gini_df['Country Name'].
unique()]),
                  dcc.Graph(id='gini_country_barchart')
          ])
      ])
    ]
])
```

5. Теперь можно создать первую функцию обратного вызова, принимающую в качестве входного параметра год и возвращающую соответствующий график. Обратите внимание, как в начале функции используется исключение `PreventUpdate`:

```
@app.callback(Output('gini_year_barchart', 'figure'),
              Input('gini_year_dropdown', 'value'))
def plot_gini_year_barchart(year):
    if not year:
        raise PreventUpdate
    df =\
        gini_df[gini_df['year'].eq(year)].sort_values(gini).
dropna(subset=[gini])
    n_countries = len(df['Country Name'])
    fig = px.bar(df,
                 x=gini,
                 y='Country Name',
                 orientation='h',
                 height=200 + (n_countries*20),
                 title=gini + ' ' + str(year))
    return fig
```

6. Мы можем сразу написать и вторую функцию обратного вызова для обработки правой части секции:

```
@app.callback(Output('gini_country_barchart', 'figure'),
                Input('gini_country_dropdown', 'value'))
def plot_gini_country_barchart(country):
    if not country:
        raise PreventUpdate
    df = gini_df[gini_df['Country Name']==country].
dropna(subset=[gini])
    fig = px.bar(df,
                 x='year',
                 y=gini,
                 title=' - '.join([gini, country]))
    return fig
```

7. Наконец, запустим наше приложение:

```
if __name__ == '__main__':
    app.run_server(mode='inline')
```

В результате должно запуститься приложение, показанное на рис. 5.8.

Теперь осталось внести сделанные изменения в наше существующее приложение. Все, что нам нужно для этого сделать, – это вставить визуальные компоненты туда, где мы хотим их видеть. Функции обратного вызова при этом могут размещаться ниже атрибута `layout`. Вы можете сделать копию приложения, которое мы создали в главе 3. Вставьте новые компоненты в виде списка между строками кода `dcc.Graph(id='population_chart')` и `dbc.Tabs`, как показано в следующем фрагменте:

```
...
dcc.Graph(id='population_chart'),
html.Br(),
html.H2('Gini Index - World Bank Data', style={'textAlign': 'center'}),
html.Br(),
dbc.Row([
    dbc.Col([
...
        dcc.Graph(id='gini_country_barchart')
    ]),
]),
dbc.Tabs([
dbc.Tab([
...
```

Используя один индикатор, мы создали два динамических графика: первый строится в ответ на выбор пользователем года, а второй – на выбор страны. Также мы познакомились с двумя видами отображения столбчатых диаграмм: вертикальным и горизонтальным, и обсудили их преимущества и недостатки.

Теперь пришло время узнать, как лучше выводить множественные диаграммы в рамках одного графического объекта. Попутно мы познакомимся с новым набором индикаторов.

Разные способы отображения столбчатых диаграмм с несколькими рядами данных

Когда нам нужно отобразить данные по разным странам за одни и те же годы, у нас есть несколько вариантов того, как можно вывести несколько столбиков для каждой категории по оси *x*. На рис. 5.9 показаны четыре варианта отображения переменных **a** и **b** на столбчатой диаграмме.

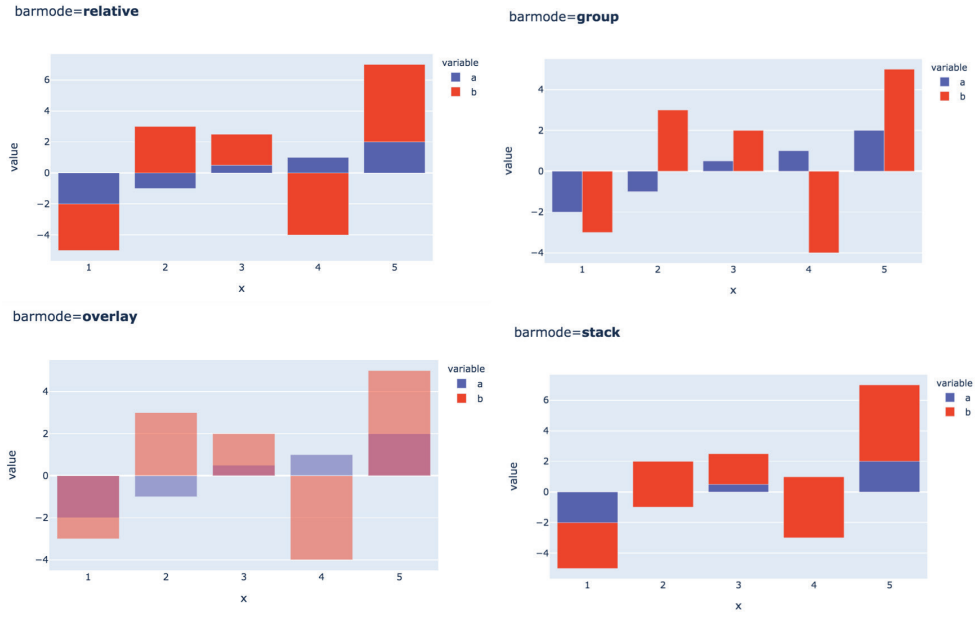


Рис. 5.9. Использование опции `barmode` при отображении данных

Датафрейм, использованный для построения этих графиков, показан ниже:

```
df = pd.DataFrame({
    'x': [1, 2, 3, 4, 5],
    'a': [-2, -1, 0.5, 1, 2],
    'b': [-3, 3, 2, -4, 5]
})
df
```

А код для их создания был использован такой:

```
for mode in ['relative', 'stack', 'group',   'overlay']:
    fig = px.bar(df, x='x', y=['a', 'b'], barmode=mode,
```

```
    title=f'barmode=<b>{mode}</b>')
        fig.show()
```

Четыре диаграммы, показанные на рис. 5.9, отображают одну и ту же информацию, но разными способами. Эти способы задаются при помощи параметра `barmode`. В случае с относительным расположением рядов данных (**`barmode=relative`**, слева вверху) столбики располагаются поверх друг друга, при этом отрицательные значения опускаются ниже осевой линии, а положительные поднимаются выше. Отображение данных с группировкой (**`barmode=group`**, справа вверху) позволяет расположить столбики рядом друг с другом. Вариант наложения (**`barmode=overlay`**, слева внизу) предполагает расположение столбиков на фоне друг друга с определенным уровнем прозрачности, а накопительный способ отображения (**`barmode=stack`**, справа внизу) похож на относительный, но отрицательные значения нивелируют положительные, что видно на примере второго и четвертого столбиков на последней диаграмме. Это бывает удобно, если вам нужно сравнить доли каждого значения в общей массе, особенно если для них есть общая сумма. Именно этим способом мы воспользуемся при отображении информации о доходах, имеющейся в нашем наборе данных.

Создание датафрейма с данными о доходах

Давайте взглянем на пять столбцов, в которых показаны данные о доходах для пяти квинтилей по численности населения. Сначала создадим поднабор данных на основе датафрейма `poverty` и сохраним его отдельно под именем `income_share_df`. Сделаем это при помощи фильтрации колонок с использованием регулярных выражений, как показано ниже. Также мы избавимся от пропущенных значений:

```
income_share_df =\
poverty.filter(regex='Country Name|^year$|Income share.*?20').dropna()
income_share_df
```

Запуск этого кода сформирует новый датафрейм и выведет его на экран, как показано на рис. 5.10.

	Country Name	year	Income share held by fourth 20%	Income share held by highest 20%	Income share held by lowest 20%	Income share held by second 20%	Income share held by third 20%
67	Albania	1996	23.3	36.1	9.2	13.7	17.7
73	Albania	2002	22.2	40.4	8.4	12.6	16.5
76	Albania	2005	22.5	39.2	8.4	12.9	17.0
79	Albania	2008	22.2	39.0	8.9	13.1	16.8
83	Albania	2012	22.8	37.8	8.9	13.2	17.3
...	...	...	...	...	...	...	...
8229	Zambia	2006	19.1	59.5	3.5	6.8	11.1
8233	Zambia	2010	17.9	61.1	3.8	6.8	10.5
8238	Zambia	2015	19.3	61.3	2.9	6.0	10.6
8279	Zimbabwe	2011	21.0	49.7	5.8	9.5	14.0
8285	Zimbabwe	2017	20.6	51.1	6.0	9.1	13.2

1674 rows × 7 columns

Рис. 5.10. Датафрейм с доходами по квинтилям населения

Для каждой комбинации из страны и года у нас есть пять показателей, каждый из которых отражает процент дохода населения из определенного квин-

тиля. Мы хотим, чтобы пользователь мог сам выбирать страну и наблюдать за тем, как менялись эти проценты по группам населения с течением лет. На рис. 5.11 показан пример такой диаграммы по Соединенным Штатам.

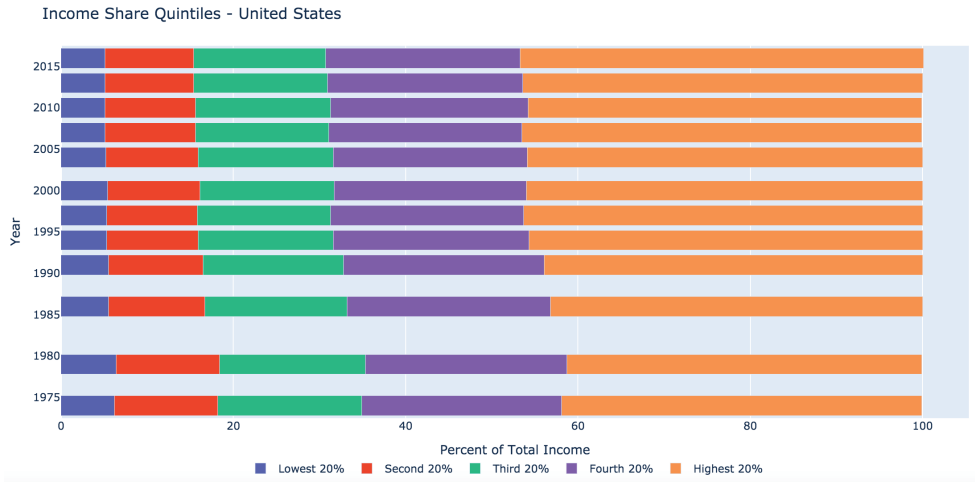


Рис. 5.11. Изменение процентов дохода по группам населения в США за последние полвека

Поскольку все значения в одном наблюдении в сумме дают 100 (с небольшой погрешностью), мы получили очень легко сравниваемые столбики, ведь их суммарная длина будет одинаковой. Здесь мы имеем дело с пропорциями, и нам интересно видеть как распределение доходов в конкретном году, так и изменение этого показателя с течением времени.

Как видите, нам легко понять, как менялись доли дохода для первого и последнего квинтилей, т. е. слева и справа на графике, поскольку эти столбики имеют одну и ту же основу, будь то их начало или конец. В то же время для остальных квинтилей уже не так просто отследить изменения, ведь вместе с длиной характеризующих их столбиков меняются и их точки отсчета. И чем больше элементов будет содержаться на графике, тем труднее будет проводить сравнение. Но, как мы знаем, диаграммы в Plotly обладают интерактивностью, и вы можете навести мышью на любые области и узнать конкретные значения.

Построить показанную выше диаграмму не составляет труда. Мы уже создали датафрейм со всеми необходимыми данными. Можно было бы просто передать значения `x` и `y`, а также установить параметр `orientation='h'`, но проблема в том, что категории в нашем датафрейме упорядочены по алфавиту, а мы хотим, чтобы они были отсортированы в соответствии с их числовыми значениями от нижней группы к верхней. Так пользователи смогут легко понять расположение столбиков на графике. Как обычно, это задача, решаемая с помощью манипуляций с данными.

1. Сначала нам нужно переименовать колонки и отсортировать их по порядку – от меньшей группы к большей. Один из способов осуществить это состоит в добавлении в начале заголовка порядкового значения с последующей сортировкой. Это можно сделать при помощи метода

rename. После этого необходимо применить метод sort_index с параметром axis=1, означающим, что мы работаем с колонками:

```
income_share_df = income_share_df.rename(columns={
    'Income share held by lowest 20%': '1 Income share held by lowest 20%',
    'Income share held by second 20%': '2 Income share held by second 20%',
    'Income share held by third 20%': '3 Income share held by third 20%',
    'Income share held by fourth 20%': '4 Income share held by fourth 20%',
    'Income share held by highest 20%': '5 Income share held by highest 20%'
}).sort_index(axis=1)
```

2. Проверим, что все сделано правильно:

```
income_share_df.columns

Index(['1 Income share held by lowest 20%',
       '2 Income share held by second 20%',
       '3 Income share held by third 20%',
       '4 Income share held by fourth 20%',
       '5 Income share held by highest 20%',
       'Country Name', 'year'],
      dtype='object')
```

3. Теперь удалим ненужный текст из заголовков. Для этого можно воспользоваться регулярными выражениями из стандартной библиотеки re. Заменим любые цифры с последующим строковым выражением "Income share held by" на пустые строки, после чего преобразуем первую букву заголовка в заглавную, как показано ниже:

```
import re
income_share_df.columns = [
re.sub('\d Income share held by ', '', col).title() for
col in income_share_df.columns
]
```

4. Создадим переменную income_share_cols для обращения к именам интересующих нас столбцов:

```
income_share_cols = income_share_df.columns[:-2]
income_share_df
Index(['Lowest 20%', 'Second 20%', 'Third 20%', 'Fourth 20%',
'Highest 20%'], dtype='object')
```

5. Теперь наш датафрейм содержит простые и понятные заголовки и готов к визуализации. Сначала создадим переменную country для фильтрации данных:

```
country = 'China'
```

6. Создадим столбчатую диаграмму с помощью функции `px.bar`. Обратите внимание, что в качестве параметра `x` мы передаем список. Plotly Express умеет работать и с данными в широком формате, что в нашем случае очень удобно. Мы могли бы отменить свертывание в нашем датафрейме, применив метод `melt`, как показывали в предыдущей главе. Также мы передадим следующие значения параметров в функцию: `orientation='h'` и `barmode='stack'`, а в заголовок динамически поместим название страны, как показано ниже:

```
fig = px.bar(income_share_df[income_share_df['Country
Name']==country].dropna(),
             x=income_share_cols,
             y='Year',
             barmode='stack',
             height=600,
             hover_name='Country Name',
             title=f'Income Share Quintiles - {country}',
             orientation='h')
```

7. Возможно, вы заметили, что я присвоил результат функции переменной `fig`. Дело в том, что нам нужно еще внести некоторые изменения. Запуск этого фрагмента кода приведет к отображению графика, показанного на рис. 5.12.

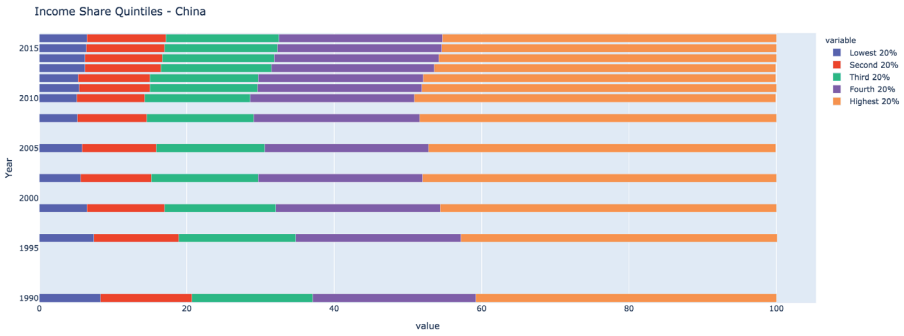


Рис. 5.12. Проценты дохода по квинтилям с опциями по умолчанию

8. Как видите, заголовки оси *x* и легенды получились малоинформативными. Сейчас мы их поменяем, а легенду перенесем под диаграмму с соблюдением логического порядка. Это поможет пользователям легче ассоциировать столбики на графике со значениями из легенды и в целом сделает визуализацию более привлекательной. Как мы уже говорили в главе 3, все свойства оформления можно задать при помощи атрибута `fig.layout`, и сделать это очень просто. Обратите внимание, что атрибут `legend` имеет вложенные атрибуты `x` и `y`, с помощью которых можно задать относительное местоположение легенды на графике. Мы установим атрибуту `x` значение 0.25, чтобы легенда начиналась с отступом в 25 % от левого края диаграммы:

```
fig.layout.legend.title = None
fig.layout.legend.orientation = 'h'
fig.layout.legend.x = 0.25
fig.layout.xaxis.title = 'Percent of Total Income'
```

9. Запуск написанного нами кода для Индонезии приведет к отображению графика, показанного на рис. 5.13.

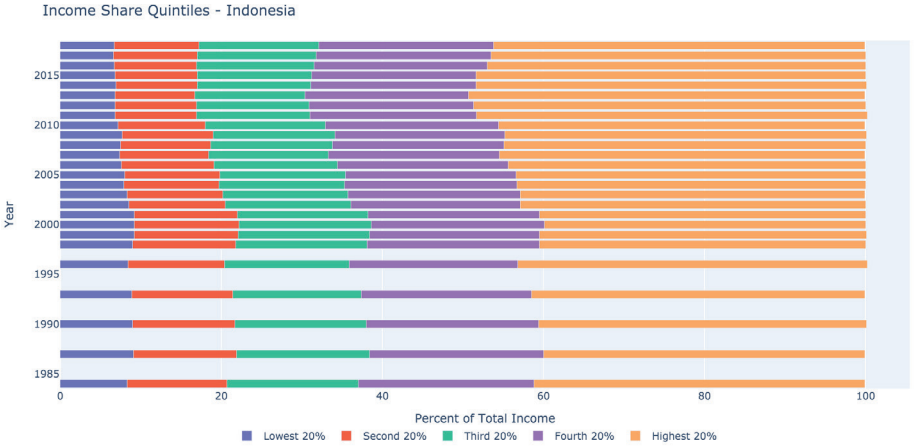


Рис. 5.13. Проценты дохода по квинтилям с измененными опциями

Теперь давайте соберем все воедино и внесем изменения в существующее приложение.

Внедрение изменений в приложение

Итак, мы опробовали все внесенные изменения и готовы внедрить их в наше приложение. Никаких подробных объяснений этот процесс более не требует, поскольку мы уже выполняли его неоднократно. Мы лишь пройдемся по самим шагам, а полный код приложения вы найдете в репозитории к книге.

1. Для начала мы объявим измененный датафрейм и внесем необходимые изменения в колонки. Убедитесь в том, что код, приведенный ниже, вставлен ниже объявления датафрейма `poverty`, поскольку он на него ссылается:

```
income_share_df = poverty.filter(regex='Country Name|^year$|Income
share.*?20').dropna()
income_share_df = income_share_df.rename(columns={
    'Income share held by lowest 20%': '1 Income share held by
lowest 20%',
    'Income share held by second 20%': '2 Income share held by
second 20%',
    'Income share held by third 20%': '3 Income share held by
third 20%',
```

```
    'Income share held by fourth 20%': '4 Income share held by
fourth 20%',
    'Income share held by highest 20%': '5 Income share held by
highest 20%'
}).sort_index(axis=1)

income_share_df.columns = [re.sub('\d Income share held by ', '',
col).title()
                           for col in income_share_df.columns]
income_share_cols = income_share_df.columns[:-2]
```

2. В макете мы добавим элемент `h2` с заголовком нашей новой секции, а также выпадающий список для стран и график – все это разместим под последним графиком с исследованием индекса Джини:

```
dbc.Row([
    dbc.Col(lg=1),
    dbc.Col([
        html.H2('Income Share Distribution', style={'textAlign':
'center'}),
        html.Br(),
        dcc.Dropdown(id='income_share_country_dropdown',
                     options=[{'label': country, 'value': country}
                              for country in income_share_
df['Country Name'].unique()]),
        dcc.Graph(id='income_share_country_barchart')
    ], lg=10)
]),
```

3. Теперь создадим функцию обратного вызова по примеру тех, которые делали до этого. Окончательный код функции показан ниже:

```
@app.callback(Output('income_share_country_barchart', 'figure'),
Input('income_share_country_dropdown', 'value'))
def plot_income_share_barchart(country):
    if country is None:
        raise PreventUpdate
    fig = px.bar(income_share_df[income_share_df['Country
Name']==country].dropna(),
                 x=income_share_cols,
                 y='Year',
                 barmode='stack',
                 height=600,
                 hover_name='Country Name',
                 title=f'Income Share Quintiles - {country}',
```

```
                orientation='h',
                )
    fig.layout.legend.title = None
    fig.layout.legend.orientation = 'h'
    fig.layout.legend.x = 0.2
    fig.layout.xaxis.title = None
    return fig
```

Если вы все сделали правильно и добавили код в нужные места приложения, все должно работать. Итак, мы добавили в наше приложение несколько индикаторов, с которыми пользователь может интерактивно взаимодействовать, и графики, отображающие аналитическую информацию в разных разрезах.

Четыре способа отображения столбчатой диаграммы – это здорово, но если мы хотим, чтобы пользователь мог добавлять в сравнение более одной страны, анализировать итоговые графики будет очень неудобно. Возвращаясь к графику по индексу Джини, стоит отметить, что для каждой страны на диаграмме выводится от 20 до 30 столбиков в зависимости от того, сколько данных доступно. Для четырех стран это число увеличится до ста, что сильно затруднит чтение визуализации.

А как насчет того, чтобы позволить пользователю выбирать столько стран, сколько он хочет, и для каждой из них строить отдельный график в виде фасета в составе единого графического объекта?

Именно это мы и сделаем в следующем разделе.

Использование ячеистой структуры для вывода множественных диаграмм (фасетирование)

Фасетирование (faceting) – это очень мощная техника, позволяющая добавить новое измерение в наш анализ. При помощи нее мы можем выбрать любой признак (столбец) из нашего датафрейма, по элементам которого хотим разбить график. Если вы ожидаете, что сейчас будет пространное описание того, как это работает и что вы должны изучить, нет, этого не будет. Как и в большинстве случаев при работе с модулем Plotly Express, если ваши данные приведены в опрятный длинный формат, все, что вам останется сделать, – это передать название колонки параметру `facet_col` или `facet_row`. Серьезно, это все!

Давайте рассмотрим доступные опции в виде параметров, имеющих отношение к фасетированию:

- `facet_col`: передача этого параметра означает, что вы хотите выводить графики в виде колонок, а указанный по имени столбец будет использоваться для их разбиения. В результате диаграммы будут располагаться бок о бок, как колонки в таблице;
- `facet_row`: смысл этого параметра такой же, как у предыдущего, но графики в этом случае будут выводиться в виде строк, т. е. один под другим;

- `facet_col_wrap`: этот параметр может оказаться полезным, если вам необходимо выводить динамическое количество графиков. Если вы даете пользователю возможность создавать множественные графики, то должны позаботиться о том, чтобы через определенное количество фасетов начиналась новая строка. Этот параметр как раз и отвечает за максимальное количество графиков в одной строке;
- `facet_row_spacing` и `facet_col_spacing`: как ясно из названий этих параметров, с их помощью вы можете управлять расстоянием между колонками и строками, в которых располагаются графики. Расстояние задается в интервале [0, 1] и соответствует процентной доле горизонтального или вертикального размера диаграммы.

Давайте приведем небольшой пример фасетирования, чтобы вам было лучше понятно, о чем идет речь.

1. Создайте список со странами для фильтра:

```
countries = ['Algeria', 'Japan']
```

2. Измените определение датафрейма `df` таким образом, чтобы в него входили только страны, присутствующие в заданном ранее списке:

```
df = gini_df[gini_df['Country Name'].isin(countries)].
dropna(subset=[gini])
```

3. Вызовите функцию `px.bar` с параметром `facet_row='Country Name'`:

```
px.bar(df,
       x='year',
       y=gini,
       facet_row='Country Name')
```

Запуск этого кода приведет к выводу комбинированного графика, показанного на рис. 5.14.

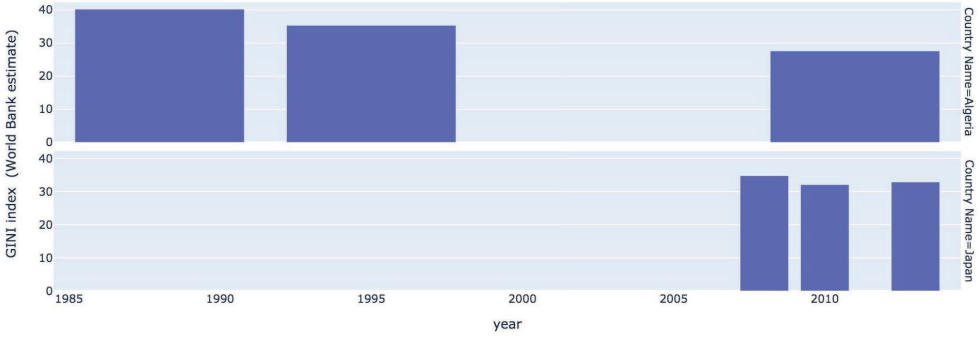


Рис. 5.14. Две столбчатые диаграммы, созданные при помощи параметра `facet_row`

4. Как видите, разбить график на несколько довольно просто, и при этом все фасеты будут подписаны индивидуальными названиями стран. Но вывод по-прежнему не отвечает всем нашим требованиям. Подписи на оси *y* накладываются друг на друга, а чтобы прочитать название страны справа от графика, придется постараться. Давайте немного поработаем с выводом. Начнем с модификации отображения подписей на вертикальной оси. Изменить их можно при помощи параметра `labels`, передав словарь с новым заголовком, как показано ниже:

```
labels={gini: 'Gini Index'}
```

5. Также мы можем облегчить пользователю чтение графиков за счет цветовой идентификации данных по разным странам. Это позволит сделать их лучше различимыми и к тому же раскрасит в те же цвета легенду. Опять же, сделать это можно, всего лишь задав параметр `color` с указанием имени столбца, который мы хотим для этого использовать:

```
color='Country Name'
```

6. Также неплохо было бы снабдить нашу визуализацию динамическим заголовком. Мы могли бы выводить полное название индикатора, а на следующей строке – список выбранных стран для анализа через запятую. Аннотации в Plotly поддерживают определенные правила разметки HTML, чем мы можем воспользоваться для перевода строки с помощью тега `<br>`:

```
title='<br>'.join([gini, ', '.join(countries)])
```

7. Две страны на графике выглядят довольно неплохо, а что, если пользователь выберет сразу семь стран? Мы можем задать динамически высоту общей диаграммы, как уже делали это в случае с визуализацией индекса Джини. Используем ту же технику, но с другими значениями, поскольку здесь речь идет о целых фасетах, а не о горизонтальной столбчатой диаграмме:

```
height = 100 + 250*len(countries)
```

8. Обновленный вызов функции будет выглядеть так, как показано ниже:

```
px.bar(df,
       x='year',
       y=gini,
       facet_row='Country Name',
       labels={gini: 'Gini Index'},
       color='Country Name',
       title='<br>'.join([gini, ', '.join(countries)]),
       height=100 + 250*len(countries))
```

9. Итоговая визуализация показана на рис. 5.15.

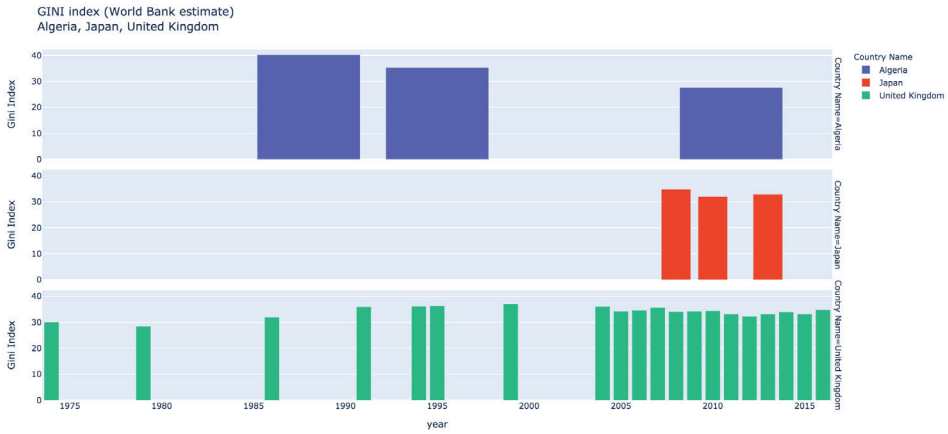


Рис. 5.15. Три столбчатые диаграммы с дополнительными опциями

В результате мы видим, что визуализация обновилась, и теперь на ней выводится столько графиков, сколько стран выбрал для просмотра пользователь. Единственное, что осталось сделать, – это изменить определение выпадающего списка таким образом, чтобы пользователь имел возможность осуществлять множественный выбор. Сейчас мы это сделаем, а заодно подумаем, как еще можно улучшить внешний вид нашего макета.

Исследование дополнительных возможностей выпадающих списков (множественный выбор, заместители текста и т. д.)

У компонента `Dropdown`, представляющего выпадающий список, есть необязательный параметр `multi`, принимающий булево значение, и если вам необходимо позволить пользователю выбирать более одного элемента, вы можете передать этому параметру значение `True`, как показано ниже:

```
dcc.Dropdown(id='gini_country_dropdown',
             multi=True,
             options=[{'label': country, 'value': country}
                      for country in gini_df['Country Name'].unique()]),
```

Теперь пользователь сможет в выпадающем списке выбрать столько стран, сколько ему нужно, и высота объединенной секции с графиками будет динамически меняться в зависимости от количества выбранных им элементов. Давайте посмотрим, насколько интуитивно понятной мы сделали эту возможность множественного выбора.

Добавление заместителя текста для выпадающего списка

Если взглянуть на рис. 5.16, на котором показан фрагмент нашего приложения с выпадающими списками, вы поймете, что внешне они ничем не отличаются.

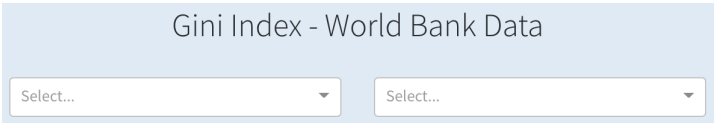


Рис. 5.16. Выпадающие списки без заместителей текста

В списках указано слово `Select`, то есть приглашение сделать выбор. А что именно нужно выбрать?

У компонента `Dropdown` есть необязательный параметр `placeholder`, с помощью которого можно подсказать пользователю, какой выбор необходимо сделать.

Давайте зададим этот параметр для обоих наших списков следующим образом:

```
placeholder="Select a year"
placeholder="Select one or more countries"
```

Можно дополнительно выделить подписи к выпадающим спискам, воспользовавшись компонентом `Label` пакета Dash Bootstrap Components. Это позволит обозначить имя списков над ними:

```
dbc.Label("Year")
dbc.Label("Countries")
```

В результате вывод будет выглядеть так, как показано на рис. 5.17.

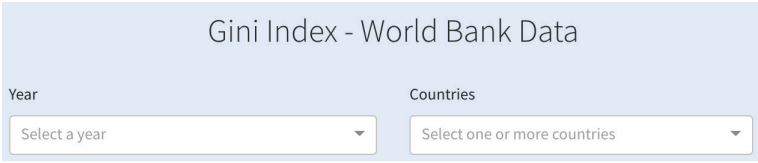


Рис. 5.17. Выпадающие списки с заместителями текста и подписями

Полагаю, теперь пользователю будет комфортнее. Как видите, с помощью заместителей текста мы указали, в каком списке необходимо сделать единичный выбор, а в каком доступен множественный. Можно применить такой же подход и в отношении секции `Income Share Distribution`, добавив подпись `Country` и заместитель `Select a country`.

С нашими последними изменениями приложение довольно сильно увеличилось в размерах, и в нем присутствует достаточно большое богатство выбора. Давайте взглянем на него и подумаем, что еще можно сделать, чтобы улучшить его внешний вид и облегчить работу с ним.

Изменение темы приложения

Мы уже показывали, как легко можно изменить тему приложения, передав список в качестве значения параметра `external_style_sheets` при создании экземпляра приложения. Например, следующим образом вы можете задать стиль `COSMO`:

```
app = dash.Dash(__name__,
                external_stylesheets=[dbc.themes.COSMO])
```

Это приведет к визуальному изменению некоторых элементов в вашем приложении.

Также вы можете адаптировать внешний вид приложения под визуальный стиль используемых графиков. Например, можно установить цвет фона приложения в соответствии с цветом по умолчанию объектов Plotly. Это можно сделать, передав параметру `style` верхнеуровневого элемента `html.Div` нужный цвет, как показано ниже:

```
app.layout = html.Div([
...
], style={'backgroundColor': '#E5ECF6'})
```

Но нужно сделать кое-что еще. Объект `Figure` содержит две основные области: `plot` и `paper`. Первая отвечает за внутренний прямоугольный контур в рамках осей *x* и *y*. На всех графиках, которые мы показывали, эта область окрашена в светло-голубой цвет.

Область `paper` представляет собой обрамляющий прямоугольник. На показанных выше рисунках эта область закрашена белым, и мы вправе задать для нее любой другой цвет. Для этого достаточно добавить в функцию обратного вызова, строящую график, следующий код:

```
fig.layout.paper_bgcolor = '#E5ECF6'
```

Если сейчас запустить приложение, мы увидим пустые графики с белым фоном для тех объектов, для которых не были заданы значения по умолчанию. Для них необходимо создать пустые объекты `Figure` с установленным цветом фона для единообразия. Сделать это очень просто, как мы уже показывали в главе 3. У компонента `dcc.Graph` есть атрибут `figure`, которому можно присвоить пустой объект `Figure` с заданным фоном. Этот объект будет меняться, когда пользователь сделает свой выбор в соответствующем выпадающем списке. Поскольку у нас в приложении не один график, а несколько, удобнее будет написать отдельную функцию, которую можно будет вызывать всякий раз, когда необходимо создать пустой график. Пример такой функции показан ниже:

```
import plotly.graph_objects as go

def make_empty_fig():
    fig = go.Figure()
    fig.layout.paper_bgcolor =  '#E5ECF6'
    fig.layout.plot_bgcolor = '#E5ECF6'
    return fig
```

Теперь мы можем добавить вызовы функции `make_empty_fig` во всех местах приложения, где в этом есть необходимость, следующим образом:

```
dcc.Graph(id='gini_year_barchart',
          figure=make_empty_fig())
```

Изменение размеров компонентов

Отдельно стоит поговорить о влиянии изменения размеров окна браузера на размер и расположение различных компонентов приложения. Объекты с графиками по умолчанию являются адаптивными, но нам необходимо определиться с некоторыми характеристиками элементов, располагающихся бок о бок. В секции с индексом Джини у нас есть два таких объекта, размещенных в своих компонентах `dbc.Col` в один ряд. Все, что нам нужно сделать в их отношении, – это задать их размер в условных колонках для большого (`lg`) и среднего (`md`) размера экрана, как показано ниже:

```
dbc.Col([
...
], md=12, lg=5),
```

В результате при открытии приложения на большом экране каждый объект будет занимать ширину в пять ячеек из 12 возможных, на которые экран разбивает Bootstrap. В главе 1 мы в общих чертах обсуждали эту технику размещения элементов на экране, используемую в библиотеке Bootstrap. На экране среднего размера графики будут занимать 12 из 12 ячеек, т. е. всю доступную ширину приложения.

Когда мы только начинали погружаться в тему интерактивности, мы создали простой график в верхней части нашего приложения, показывающий численность населения выбранной страны за 2010 год. Теперь мы можем избавиться от этой визуализации, поскольку она обладает весьма ограниченным функционалом. Это можно сделать, удалив соответствующий компонент с областью вывода под ним, а также обслуживающую его функцию обратного вызова.

В результате работы, проделанной в этой главе, внешний вид вашего приложения должен оказаться таким, как на рис. 5.18.

Я настоятельно рекомендую вам вносить все изменения в приложение вручную, без использования кода из репозитория. Также будет полезно, если вы попутно будете проверять другие доступные варианты и опции, несмотря на ошибки, которые неизбежно будут возникать в процессе.

Многократное внесение изменений в приложение с удерживанием кода под вашим полным контролем требует от вас последовательного и понятного именования компонентов, функций и переменных. Также будет полезно логически организовывать свои компоненты. Мы еще не раз будем об этом говорить и надеемся, что в результате вы выработаете такую привычку.

После всех произведенных изменений наше приложение стало выглядеть лучше, и использовать его стало удобнее. Им можно даже поделиться с друзьями и коллегами. Мы немало сделали, чтобы вывести наше приложение на новый качественный уровень, и теперь пришло время вспомнить, что нового мы узнали.

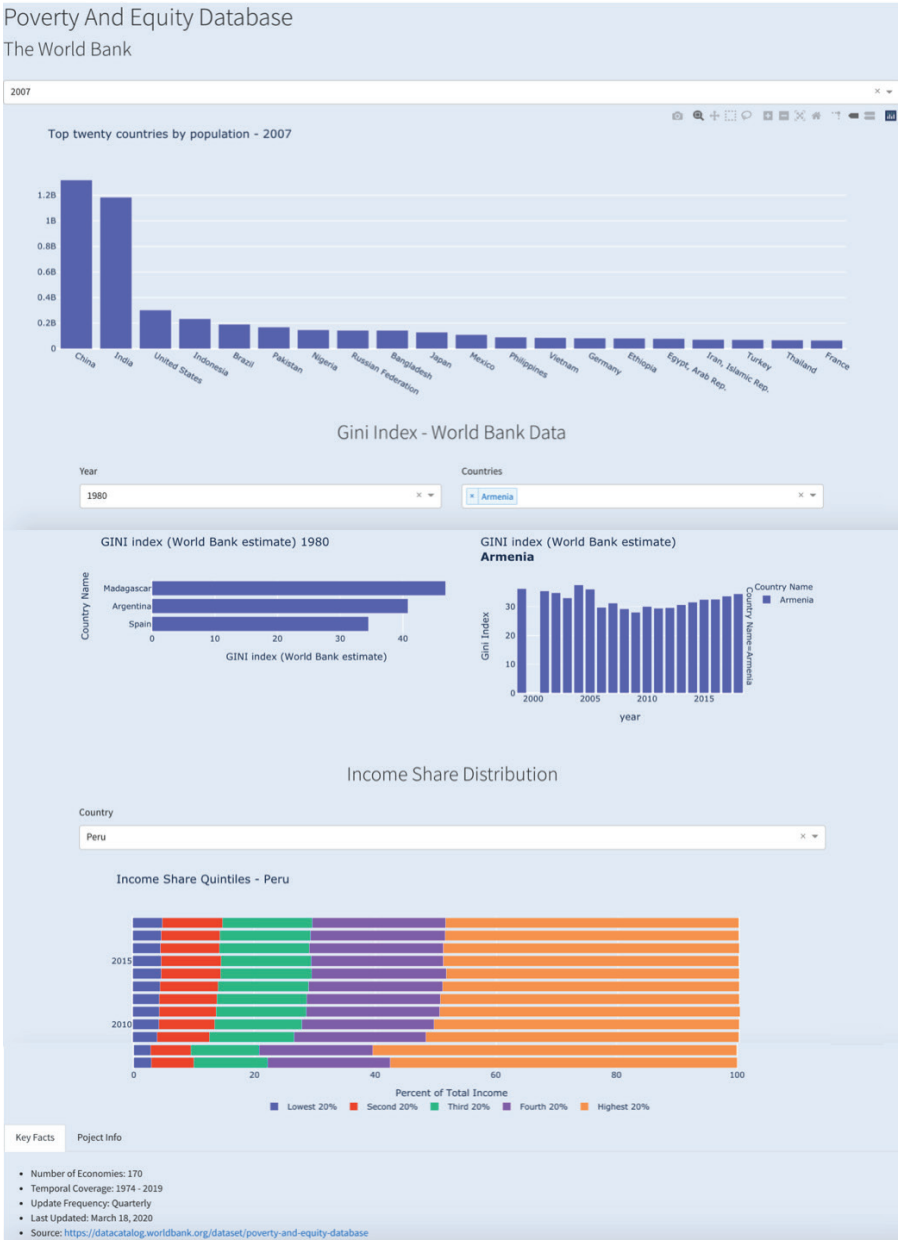


Рис. 5.18. Итоговый вид приложения

Заключение

Главным образом мы в этой главе говорили о столбчатых диаграммах, совместно с которыми активно использовали выпадающие списки. В процессе изучения мы обсудили преимущества и недостатки горизонтальной и верти-

кальной ориентации столбчатой диаграммы и внедрили оба варианта в своем приложении. После этого мы рассмотрели различные способы отображения столбчатых диаграмм с несколькими рядами данных и реализовали один из них с выводом долей от целого. Далее мы затронули тему фасетов и увидели, насколько они являются более гибкими и масштабируемыми. Также мы связали эти визуализации с выпадающими списками, что позволило пользователям осуществлять множественный выбор элементов. Убедившись, что приложение работает так, как мы и задумывали, мы сделали его более привлекательным за счет установки темы и настройки цвета фона графических элементов. Кроме того, мы установили различные размеры элементов приложения в зависимости от характеристик экрана используемого устройства. В качестве вишенки на торте мы снабдили элементы приложения подписями и аннотациями, после чего сделали скриншот нашего приложения.

В следующей главе мы подробнее поговорим об одном из самых популярных видов диаграмм – точечной. Совместно с ней мы будем использовать слайдеры, с помощью которых пользователь сможет выбирать и изменять значения или их диапазоны на графике.

Глава 6

Исследование переменных при помощи точечной диаграммы и фильтрация наборов данных

В этой главе мы рассмотрим примеры использования *точечной диаграммы* или *диаграммы рассеяния* (scatter plot) – одного из самых универсальных, полезных и широко используемых типов визуализации. В соответствии с названием диаграммы мы будем рассеивать маркеры (которые могут быть представлены квадратами, окружностями, точками, пузырями или иными символами) на координатной плоскости, а значение, которое они представляют, будет выражаться посредством их проекций на осях. Другие визуальные атрибуты, такие как размер, цвет или символы, могут быть использованы для выражения прочих характерных особенностей точек данных, как мы уже видели ранее. Поскольку все фундаментальные идеи относительно построения графиков мы уже обсудили в предыдущих главах, здесь мы не будем снова подробно на них останавливаться, а в основном сконцентрируемся на опциях, доступных при создании и настройке точечных диаграмм. Также мы тесно поработаем со слайдерами или ползунками как еще одним примером интерактивных компонентов.

Давайте приступим, но для начала перечислим темы, которые будут рассмотрены в главе:

- различные способы использования точечных диаграмм: маркеры, линии и текст;
- отображение нескольких рядов данных на одной точечной диаграмме;
- настройка цветов на точечной диаграмме;
- управление наложениями и выбросами при помощи прозрачности, символов и масштаба;
- знакомство со слайдерами, включая слайдеры диапазонов;
- настройка подписей и значений слайдеров.

Технические требования

В данной главе мы продолжим использовать те же инструменты и пакеты, что и в предыдущих. При создании точечных диаграмм мы будем использовать модуль `graph_objects`, поскольку он предлагает массу вспомогательных возможностей для настройки графиков. Пакеты, которые мы будем использовать: Plotly, Dash, Dash Core Components, Dash HTML Components, Dash Bootstrap Components, pandas и JupyterLab.

Исходный код к этой главе располагается в хранилище GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_06**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3ancblu**.

Давайте начнем с описания того, что именно мы можем визуализировать при помощи точечной диаграммы или диаграммы рассеяния.

Различные способы использования точечных диаграмм: маркеры, линии и текст

При использовании модуля `graph_objects` у вас есть масса возможностей для создания и настройки точечных диаграмм, так что мы будем использовать его совместно с Plotly Express. Чтобы вы поняли, насколько разнообразными могут быть точечные диаграммы, давайте извлечем все методы объекта `Figure`, в которых присутствует слово `scatter`, и то же самое проделаем с модулем Plotly Express:

```
import plotly.graph_objects as go
import plotly.express as px

fig = go.Figure()
[f for f in dir(fig) if 'scatter' in f]
['add_scatter',
 'add_scatter3d',
 'add_scattercarpet',
 'add_scattergeo',
 'add_scattergl',
 'add_scattermapbox',
 'add_scatterpolar',
 'add_scatterpolargl',
 'add_scatterternary']

[f for f in dir(px) if 'scatter' in f]
['scatter',
 'scatter_3d',
 'scatter_geo',
```

```
 'scatter_mapbox',
 'scatter_matrix',
 'scatter_polar',
 'scatter_ternary']
```

Как видите, в этих двух модулях есть как пересекающиеся методы, так и характерные только для одного из них. Мы не будем детально останавливаться на каждом из них, но знать об их существовании и возможности их использования при необходимости вам будет полезно. Рассмотрим разные опции при создании точечной диаграммы.

Маркеры, линии и текст

Одним из наиболее полезных параметров объекта `go.Scatter` является `mode`. Он может принимать любые комбинации значений, указывающие на вывод маркеров, линий и/или текста. При этом в комбинации можно указать одну, две или даже все три опции вместе. Указывая более одного элемента, необходимо отделять их друг от друга знаком + (плюс), например `"markers+text"`. Давайте сначала познакомимся с индикаторами, которые будем выводить в этой главе, а затем рассмотрим варианты имеющихся в нашем распоряжении опций.

1. Импортируем указанные ниже пакеты и создадим датафрейм с именем `poverty`:

```
import pandas as pd
import plotly.graph_objects as go
poverty = pd.read_csv('data/poverty.csv')
```

2. В нашем наборе данных содержатся сведения о трех категориях дневного дохода, по которым оценивается уровень бедности. Первый показатель характеризует средний дефицит дохода или потребления от черты бедности на уровне $1,90 в день. Оставшиеся два – на уровне $3,20 и $5,50 соответственно. Есть и колонки с абсолютными величинами, но мы будем работать с процентами. Имена нужных нам столбцов начинаются с `Poverty gap`, что можно использовать в шаблоне при их поиске следующим образом:

```
perc_pov_cols = poverty.filter(regex='Poverty gap').columns
perc_pov_cols
Index(['Poverty gap at $1.90 a day (2011 PPP) (%)',
       'Poverty gap at $3.20 a day (2011 PPP) (% of population)',
       'Poverty gap at $5.50 a day (2011 PPP) (% of population)'],
      dtype='object')
```

3. Для простоты выделим имена нужных нам колонок в отдельные переменные, которые будут начинаться с `perc_pov_`, чтобы было понятно, с какими показателями мы имеем дело. Не забывайте о том, что в нашем

приложении уже есть какие-то объекты и функции и необходимо стараться соблюдать простые и понятные принципы именования объектов. Используем созданный ранее список для объявления трех переменных:

```
perc_pov_19 = perc_pov_cols[0]
perc_pov_32 = perc_pov_cols[1]
perc_pov_55 = perc_pov_cols[2]
```

4. Как обычно, нам нужно взглянуть на описание индикаторов, с которыми мы собираемся работать, и, что более важно, на возможные ограничения:

```
series[series['Indicator Name']==perc_pov_19]['Short definition'][25]
```

5. Описания и ограничения для трех наших индикаторов очень похожи на те, которые мы видели в предыдущей главе. Здесь есть предостережение на предмет сравнения показателей и объективности сделанных выводов. Теперь создадим переменную для страны и используем ее и переменную `perc_pov_19` для ограничения нашего датафрейма:

```
country = 'China'
df = poverty[poverty['Country Name']==country][['year',
perc_pov_19]].dropna()
```

6. Создадим объект `Figure` и добавим к нему точечную диаграмму с использованием подходящего метода. В качестве параметра `mode` должна передаваться комбинация значений, характеризующих вывод на графике, о чем мы говорили выше, а сейчас мы просто передадим заглушку с именем `mode`:

```
fig = go.Figure()
fig.add_scatter(x=df['year'],
                y=df[perc_pov_19],
                text=df[perc_pov_19],
                mode=mode)
fig.show()
```

На рис. 6.1 и 6.2 показаны результаты запуска этого кода со всеми возможными сочетаниями опций в параметре `mode`. Использованная комбинация выводится в заголовке графика.

В модуле Plotly Express предусмотрены разные функции для создания точечных и линейных диаграмм. Вы можете выводить текст с использованием функции `scatter`, либо указав столбец в датафрейме, содержащий текстовые метки, либо передав список текстовых элементов. Для этого в функции `scatter` есть специальный параметр `text`.

Теперь посмотрим, как можно выводить информацию сразу о нескольких показателях на одной точечной диаграмме.

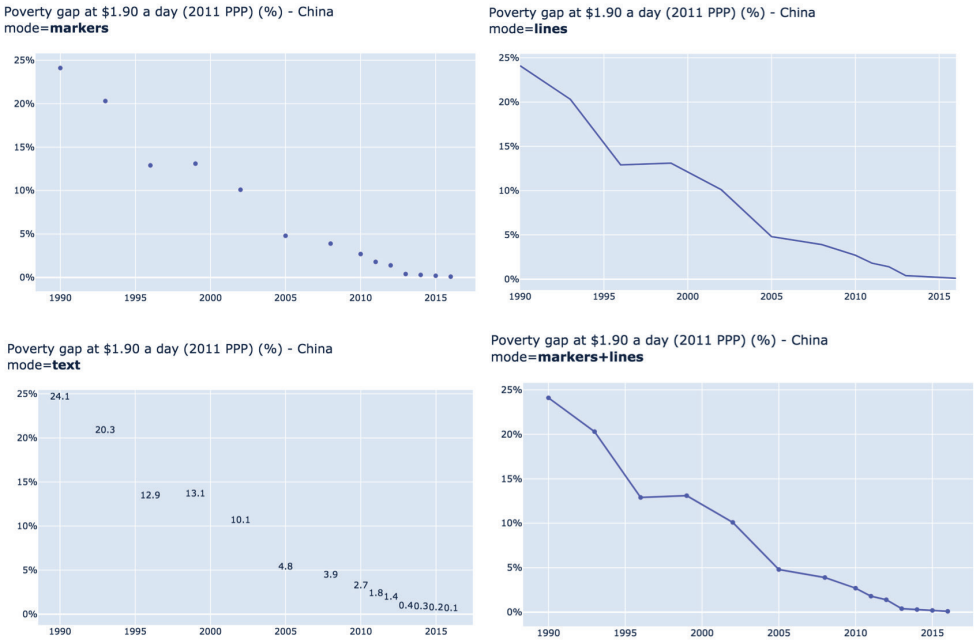


Рис. 6.1. Варианты задания параметра mode

Рис. 6.2. Варианты задания параметра mode

Отображение нескольких рядов данных на одной точечной диаграмме

В основном мы будем работать с модулем Plotly Express из-за его удобства и других преимуществ, описанных в главе 4. Но вам также очень важно уметь грамотно обращаться с объектом `Figure`, поскольку вам может понадобиться работать и с ним, особенно если предстоит выполнять массу различных настроек. К тому же не стоит забывать, что в Plotly Express реализовано хоть и большинство видов диаграмм, но далеко не все.

Давайте расширим предыдущий график за счет показателей по другим странам и сравним два подхода. Начнем с объекта `Figure` из модуля `graph_objects`.

1. Объявим список стран для анализа:

```
countries = ['Argentina', 'Mexico', 'Brazil']
```

2. Создадим поднабор данных на основе датафрейма `poverty`, который назовем `df`. В нем мы оставим только колонки `year`, `Country Name` и `perc_pov_19`, при этом значения в столбце `Country Name` должны находиться в ранее объявленном списке `countries` (отфильтруем их с помощью метода `isin`). Попутно избавимся от пропущенных значений:

```
df = (poverty
      [poverty['Country Name'].isin(countries)]
      [['year','Country Name', perc_pov_19]]
      .dropna())
```

3. Теперь создадим объект `Figure` и присвоим его переменной `fig`:

```
fig = go.Figure()
```

4. Осталось добавить ряды данных для всех стран, которые нас интересуют. Это можно сделать в цикле по странам с созданием датафреймов, в которых будет присутствовать информация только по нужной стране:

```
for country in countries:
    df_country = df[df['Country Name']==country]
```

5. В рамках того же цикла добавим на диаграмму новый ряд данных с помощью метода `add_scatter`. Заметьте, что мы передали этому методу параметр `mode='markers+lines'` и использовали параметр `name` для задания имени ряда данных в легенде:

```
fig.add_scatter(x=df_country['year'],
                y=df_country[perc_pov_19],
                name=country, mode='markers+lines')
```

6. Также нам нужно указать подпись для оси *y*, после чего можно выводить график на экран:

```
fig.layout.yaxis.title = perc_pov_19
fig.show()
```

На рис. 6.3 показан результат запуска этого фрагмента кода.

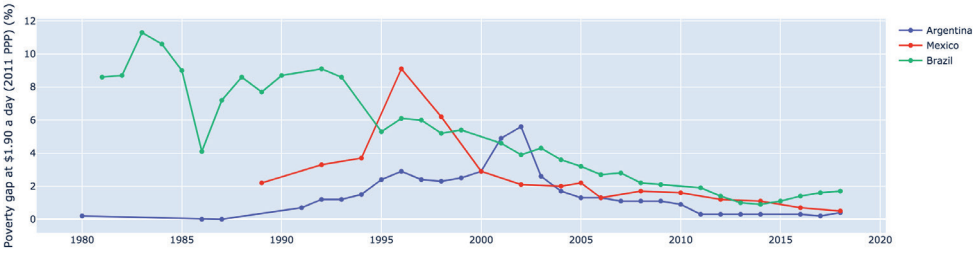


Рис. 6.3. Создание точечной диаграммы с несколькими рядами данных с помощью модуля `graph_objects`

Теперь сравним этот подход с методом, применяемым в модуле Plotly Express. Код для создания такого же графика оказался настолько коротким и интуитивно понятным, что вряд ли нуждается в дополнительных пояснениях:

```
px.scatter(df, x='year', y=perc_pov_19, color='Country Name')
```

Мы передали методу параметр `data_frame` и дополнительно указали, какие столбцы из датафрейма `df` должны быть переданы в виде значений параметрам `x`, `y` и `color`. Построенный в результате график показан на рис. 6.4.

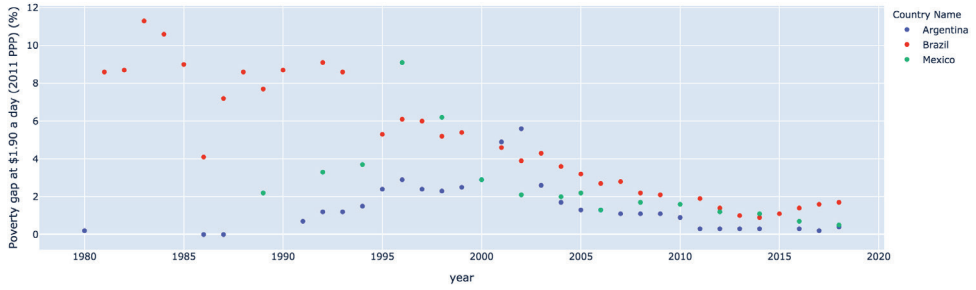


Рис. 6.4. Создание точечной диаграммы с несколькими рядами данных с помощью Plotly Express

Обратите внимание, что подписи для осей были установлены автоматически, как и оформление легенды. Более того, заголовок для легенды также указан правильно – он взят из имени колонки, переданной параметру `color`.

Но есть небольшая проблема. Разрозненные точки на графике не так просто сложить воедино – с линиями было удобнее. Это особенно критично в данном случае, поскольку здесь мы имеем дело с ярко выраженными последовательностями, и читать такие диаграммы гораздо легче с линиями. А если мы собираемся сделать нашу визуализацию интерактивной, то должны позаботиться о ясности графика при любом выборе пользователя. Как мы уже говорили, в модуле Plotly Express для точечной и линейной диаграмм предусмотрены разные

функции, так что для имитации графика с опциями `"lines+markers"` нам сначала придется построить точечную диаграмму, а затем добавить линии. Ниже приведены шаги, которые необходимо выполнить для реализации этого плана.

1. Создадим объект `Figure` и присвоим его переменной `fig`:

```
fig = px.scatter(df,
                 x='year',
                 y=perc_pov_19,
                 color='Country Name')
```

2. Теперь создадим очень похожий объект `Figure`, но другого типа:

```
fig_lines = px.line(df,
                    x='year',
                    y=perc_pov_19,
                    color='Country Name'
```

3. Сейчас нам нужно добавить ряды данных из объекта `fig_lines` в `fig`. Если вы помните, к ним у нас есть доступ посредством атрибута `data` объекта `Figure`. Атрибут `data` представляет собой кортеж, в котором в виде элементов собраны ряды данных. Таким образом, нам необходимо пройти по всем рядам данных в цикле и добавить их к объекту `fig`:

```
for trace in fig_lines.data:
    trace.showlegend = False
    fig.add_trace(trace)
fig.show()
```

Заметьте, что по умолчанию каждый добавленный на график ряд данных будет попадать в легенду. Так что нам нужно их явным образом исключить из нее. Для этого мы передали атрибуту `showlegend` значение `False`. Запуск этого кода приведет к выводу графика, показанного на рис. 6.5.

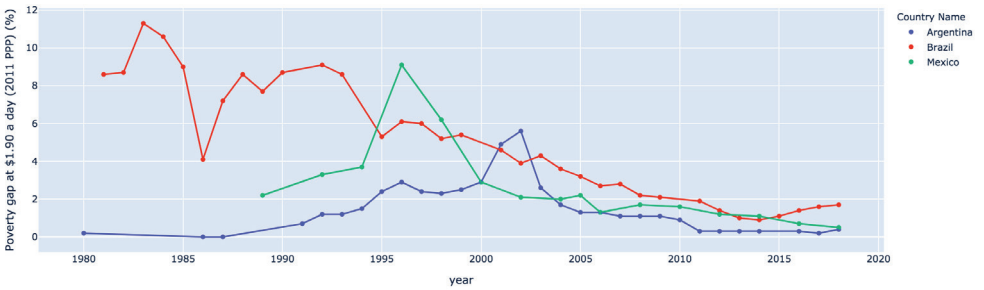


Рис. 6.5. Создание точечной диаграммы с линиями и несколькими рядами данных с помощью Plotly Express

Сравнивая усилия и объем кода, которые потребовались для создания этого графика с использованием разных подходов, можно прийти к выводу о том,

что большой разницы нет. Так часто бывает, когда вам необходимо построить диаграмму с дополнительными настройками для дальнейшей публикации. Но для быстрого старта в реализации какой-либо идеи всегда лучше подойдет Plotly Express. А уже после, когда вы определитесь с полнотой идей, которые хотите воплотить на графике, вы сможете выбрать любой подход.

В данном примере мы стали свидетелями неявного управления цветом рядов данных на графике. Эти цвета выбирались автоматически и без нашего участия. Теперь же мы хотим влиять на этот важнейший аспект визуализации. Посмотрим, какие инструменты у нас для этого есть.

Настройка цветов на точечной диаграмме

Как мы уже отметили, цветовая составляющая является очень важным фактором при выводе информации в визуальном виде. К тому же сама эта тема невероятно обширная и в своем полном объеме выходит за рамки данной книги. Мы обсудим лишь установку цветов для двух типов переменных: дискретных и непрерывных. Вместе с тем рассмотрим два способа использования цвета на графиках: установку соответствий между переменными и цветами и ручную настройку цветов.

Начнем с описания различий между двумя типами переменных.

Дискретные и непрерывные переменные

Если говорить коротко, *непрерывными переменными* (continuous variables) называются такие, которые могут принимать любые возможные значения в определенном диапазоне. К примеру, численность населения является непрерывной переменной, поскольку теоретически может быть любой. Непрерывные переменные обычно представлены числовыми значениями (целочисленными или вещественными). Высота, ширина, скорость – все это примеры непрерывных переменных.

Дискретные переменные (discrete variables), в свою очередь, могут принимать значения из заранее объявленного списка. Здесь важно уяснить, что такие переменные не могут принимать значения, находящиеся в промежутках между имеющимися в списке. Один из таких примеров – страны. Вы можете выбрать одну или другую страну, но не 10 % первой и 90 % второй. Дискретные переменные чаще всего представлены текстовыми значениями и обычно могут быть выбраны из достаточно ограниченного списка уникальных значений.

Применительно к описанным типам переменных цветовая составляющая применяется следующим образом:

- с непрерывными переменными используется цветовая шкала, значения на которой соответствуют определенному градиенту между двумя или более цветами. К примеру, если цветовая шкала начинается с белого цвета (для минимального значения) и заканчивается синим (для максимального), все значения между этими двумя крайними точками будут представлять собой градиенты разной степени от белого цвета к синему. И чем ближе цвет к синему, тем большее значение представляет

переменная в заданном диапазоне и наоборот. Совсем скоро вы увидите, как это работает;

- дискретные переменные характеризуются конкретными значениями, и заданные для них цвета должны существенно отличаться друг от друга, особенно для значений, находящихся рядом. Начнем мы с непрерывных переменных, а после этого рассмотрим и дискретные.

Использование цветов с непрерывными переменными

Будем использовать ту же метрику, с которой начали работать, и для произвольного года выведем значение индикатора по каждой стране. Мы уже знаем, как это делается. Но на этот раз добавим на график новое измерение. При помощи цвета мы продемонстрируем значение другого индикатора, например численности населения. Это позволит нам увидеть, существует ли корреляция между численностью населения и метрикой, которую мы анализируем (средний дефицит дохода или потребления от черты бедности на уровне $1,90 в день). Давайте подготовим наши переменные и данные.

1. Создадим переменные для индикатора и года:

```
indicator = perc_pov_19
year = 1991
```

2. Используя эти переменные, создадим поднабор данных на основе датафрейма `poverty`, в котором значение в столбце `year` будет соответствовать определенной нами переменной и флаг `is_country` будет равен `True`. После этого избавимся от пропущенных значений и отсортируем данные по индикатору:

```
df = poverty[poverty['year'].eq(year) & poverty['is_country']].
dropna(subset=[indicator]).sort_values(indicator)
```

3. Все, что нам нужно сделать, – это выбрать колонку, которую мы хотим ассоциировать с цветом, и это так же просто, как и все остальное в Plotly Express:

```
px.scatter(df,
           x=indicator,
           y='Country Name',
           color='Population, total')
```

Этот код сгенерирует график, показанный на рис. 6.6.

По сути, мы добавили новый слой на нашу визуализацию с выбранным столбцом. Каждый визуальный атрибут добавляет новое измерение на диаграмму, тем самым ее обогащая. В то же время чрезмерное количество измерений может существенно затруднить чтение графика. В каждом отдельном случае необходимо находить баланс между полнотой представленной информации и легкостью чтения диаграммы.

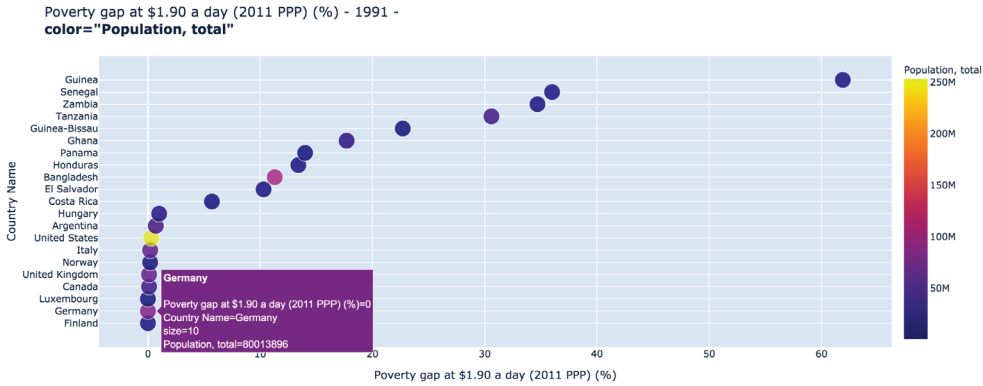


Рис. 6.6. Установка цветов для непрерывной переменной с Plotly Express

На представленном графике видно, что в стране с наибольшей численностью населения (США, светло-желтый маркер) наблюдается один из самых низких показателей нашего индикатора. А тот факт, что все остальные маркеры по цвету приближаются к темно-синему, говорит о том, что в стране с наибольшим населением значение этой метрики просто зашкаливает в сравнении с остальными странами. Таким образом, хоть по численности населения США и представляют собой выброс в этом списке, по анализируемой метрике это не так. Всплывающая подсказка, появляющаяся при наведении на маркер мышью, окрашена в тот же цвет, что облегчает задачу определения цвета выбранного маркера. *Цветовая шкала* (color scale), которую мы здесь использовали, является одной из десятков доступных вам шкал. Изменить шкалу также не составляет труда. Для этого нужно просто передать имя палитры параметру `color_scale_continuous`. На рис. 6.7 показан внешний вид диаграммы с цветовой шкалой **cividis**.

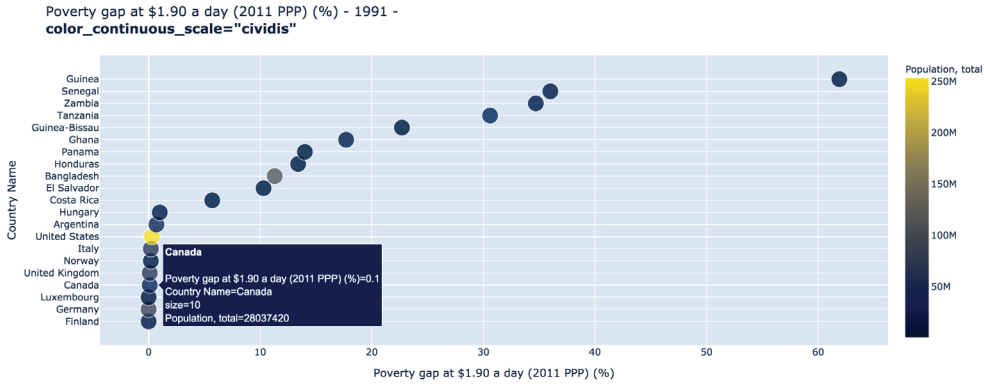


Рис. 6.7. Выбор альтернативной цветовой шкалы для непрерывной переменной

На этом графике нет никакой дополнительной информации, мы просто сменили цветовую шкалу. Но при этом всем все понятно, поскольку на шкале справа отчетливо видно, как распределяются значения по цветам. Такая шкала еще называется *последовательной* (sequential), поскольку она показы-

вает, как значения меняются от одного цвета к другому. Вы можете получить полный список имен цветовых шкал, воспользовавшись функцией `px.colors.named_colorscales()`. Но гораздо удобнее бывает сравнить цветовые шкалы наглядно и выбрать из них наиболее подходящую. Это можно сделать с помощью функции `px.colors.sequential.swatches()`, частичный вывод которой показан на рис. 6.8.

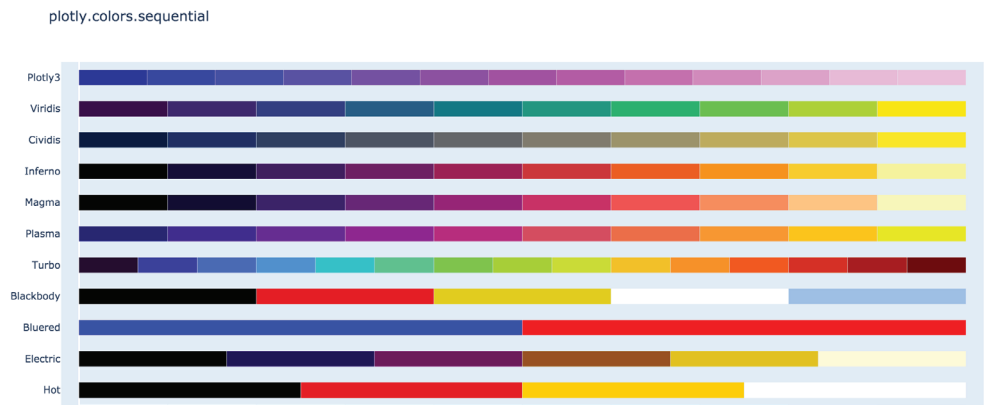


Рис. 6.8. Первые несколько цветовых шкал, доступные в Plotly

Кроме того, для демонстрации цветовых шкал вы можете воспользоваться функциями `swatches_continuous`. К примеру, на рис. 6.9 показан результат вызова функции `px.colors.sequential.swatches_continuous()`:

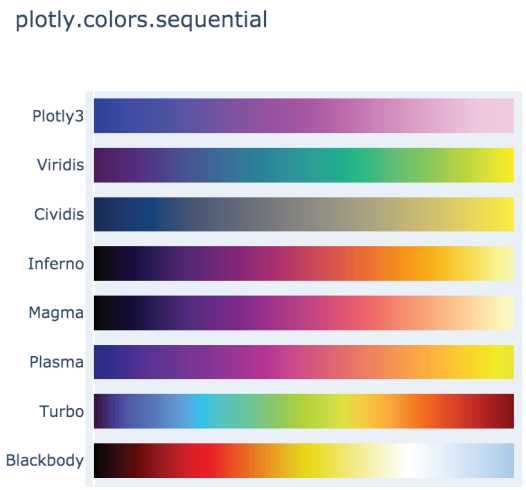


Рис. 6.9. Первые несколько цветовых шкал, доступные в Plotly, в виде цветных полосок

Здесь лучше видно, как именно выбранная шкала будет выглядеть на графиках, к тому же в таком виде целиком отображается цветовой градиент.

Вы можете использовать функции `swatches` для вывода и других типов цветовых шкал и последовательностей. Для этого достаточно вызвать функцию из

предыдущего примера, но заменить слово `sequential` на одно из следующих: `carto`, `cmocean`, `colorbrewer`, `cyclical`, `diverging` или `qualitative`.

До сих пор мы автоматически устанавливали соответствие между списком значений и цветами путем выбора нужного столбца с данными. Но можно задавать цветовые шкалы и в ручном режиме.

Создание цветовых шкал вручную

Один из способов ручного создания цветовой шкалы заключается в передаче списка из двух или более цветов параметру с именем `color_continuous_scale`. По умолчанию первый переданный вами цвет будет закреплен за минимальным значением показателя, а последний – за максимальным. Значения между ними будут окрашиваться в промежуточные градиентные цвета, при этом чем ближе значение к одному из пределов, тем ближе к граничному цвету шкалы будет и цвет. Позже мы покажем пример использования более двух цветов, а пока посмотрите на рис. 6.10, на котором выведен график с цветовой шкалой, заданной следующим образом: `color_continuous_scale=["steelblue", "darkorange"]`.

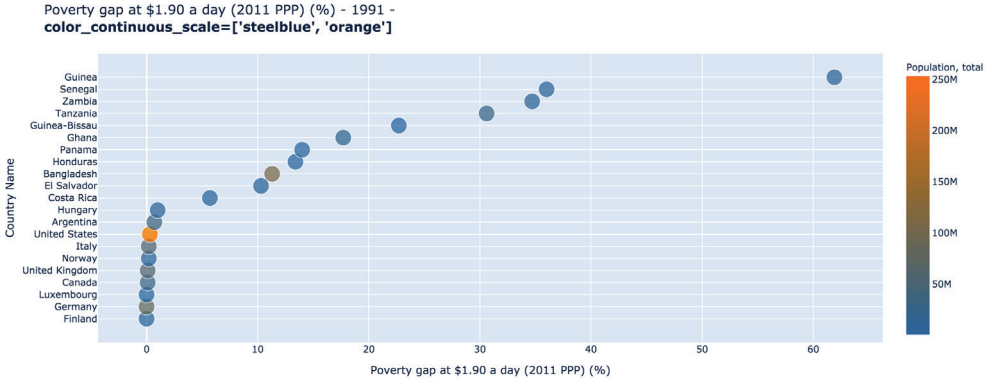


Рис. 6.10. Ручная настройка цветовой шкалы

Вы видите, как много у вас есть возможностей для управления цветом, причем это еще даже не вершина айсберга. Иногда вам необходимо перемасштабировать цветовую шкалу таким образом, чтобы градиент распространялся не так, как по умолчанию. Примером может служить рассмотренный ранее случай. На шкале, отражающей численность населения, у нас появился лишь один выброс. Так, может, лучше было передать в качестве параметра `color` масштабированную версию наших данных? В большинстве случаев рекомендуется пользоваться готовыми и хорошо протестированными цветовыми шкалами вместо создания своих вручную. Здесь вступает в игру также фактор различимости разных цветов людьми с ограниченными возможностями. Вряд ли вам хотелось бы, чтобы некоторые ваши пользователи не отличали на диаграмме один цвет от другого. В интернете можно найти таблицы с цветами, плохо воспринимаемыми людьми с ограничениями.

Теперь давайте зададим шкалу из трех цветов. Шкала `RdBu` (*red-blue* – красно-синяя) содержит на границах красный и синий цвета, а переход между ними

осуществляется через белый цвет. Это одна из цветовых шкал по умолчанию. Нарисуем простой график с использованием этой шкалы:

```
y = [-2, -1, 0, 1, 2, 3, 4, 5, 6]
px.scatter(x=range(1, len(y)+1),
           y=y,
           color=y,
           color_continuous_scale='RdBu')
```

Мы создали список целых чисел в диапазоне [–2, 6] и поставили их в соответствие цветам со шкалы `RdBu`, в результате чего получили график, показанный на рис. 6.11.

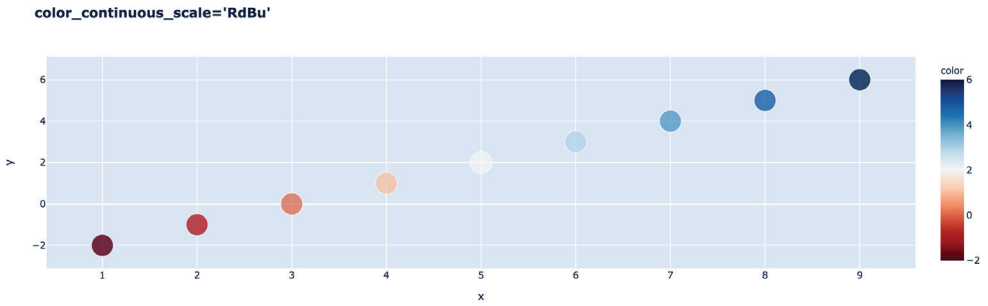


Рис. 6.11. Ручная настройка непрерывной расходящейся цветовой шкалы

Как видите, цвета маркеров постепенно меняются с красного на синий, градиентно проходя через все оттенки этих цветов. Это также называется *расходящейся* (diverging) цветовой шкалой. Есть на этой шкале и центральная точка (в данном случае белого цвета), в которой и происходит расхождение. Обычно такая точка используется для демонстрации раскола значений на две группы. Мы же хотели с помощью белой точки отделить положительные числа от отрицательных, но явно не достигли цели. В результате серединная точка разместилась точно в центре набора данных, в точке со значением 2, являющейся пятой в нашем ряду.

Это можно исправить, используя параметр `color_continuous_midpoint`, как показано на рис. 6.12.

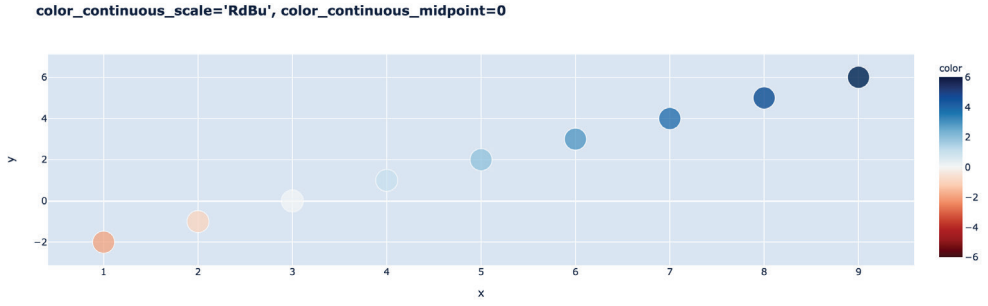


Рис. 6.12. Ручная настройка центральной точки расходящейся цветовой шкалы

Теперь, как видите, наша центральная точка белого цвета четко отделяет положительные от отрицательных чисел по вертикальной оси. Также важно, что эта точка показывает, насколько скошен наш набор данных в сторону положительных значений. Обратите внимание и на то, что на нашем графике нет ни одной красной точки. Есть пара бледно-розовых, тогда как на другом конце спектра все в порядке – темно-синие точки присутствуют. Все становится понятно при взгляде на вертикальную цветовую шкалу справа от диаграммы. Мы и близко не добрались по числам к граничному значению –6, в то время как на другой границе крайнее значение было достигнуто. Этим объясняется такой разброс цветов.

В вашем распоряжении есть и другие опции для установки цветов, масштабирования данных и выражения разных значений. Plotly предоставляет богатые возможности в этой области, и вы можете изучить их самостоятельно.

Теперь давайте посмотрим, как цветовое наполнение работает совместно с дискретными переменными.

Использование цветов с дискретными переменными

В случае с дискретными переменными мы будем использовать цвет для разбиения точек данных на определенные категории. Будем группировать значения по заданному критерию и визуализировать различия внутри групп. Сделать это очень просто – достаточно передать в качестве значения параметру `color` столбец с категориальными текстовыми значениями. К примеру, можно указать `color="Income Group"`, в результате чего будет построен график, показанный на рис. 6.13.

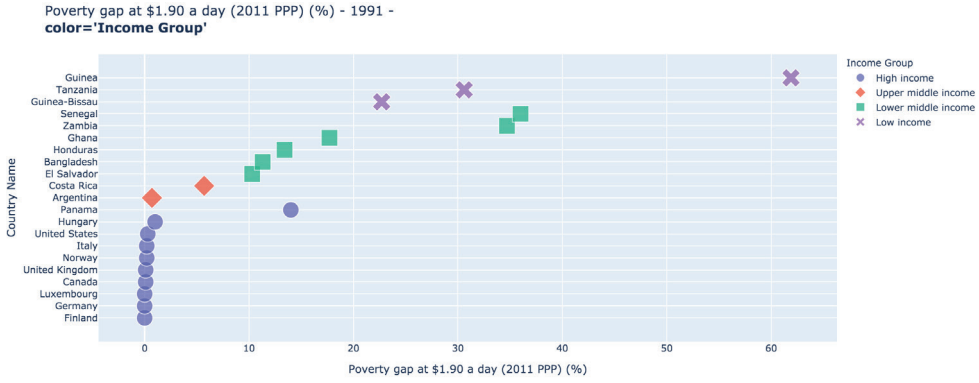


Рис. 6.13. Использование параметра `color` с категориальной переменной

Все произошло само собой. Мы просто передали параметру `color` нужный столбец, а Plotly Express сгруппировал данные по нему и выбрал цветовую палитру, с помощью которой отразил различия точек данных внутри групп. Одновременно с цветовым наполнением мы использовали и символы для внесения различий в группы. Для этого мы добавили в вызов функции следующий параметр: `symbol='Income Group'`.

Как и в случае с непрерывными переменными, с дискретными мы также можем устанавливать свою последовательность цветов для отображения с

помощью параметра `color_discrete_sequence`. На рис. 6.14 показан результат присвоения этому параметру одной из предустановленных цветовых последовательностей из Plotly.

Рис. 6.14. Установка цветовой последовательности для категориальной переменной

В данном случае мы выбрали цветовую последовательность `px.colors.qualitative.G10` из доступного нам списка, а все содержимое этого списка можно получить, воспользовавшись функцией `px.colors.qualitative.swatches()`.

Также подобно тому, как мы делали это с непрерывными переменными, мы можем вручную задавать перечень цветов для отображения дискретных переменных. Кроме того, для этого можно использовать шестнадцатеричные коды цветов вида `#aeae14` или значения RGB, например `'rgb(25, 85, 125)'`. Передав список цветов в функцию в виде параметра `color_discrete_sequence`, вы получите график, показанный на рис. 6.15.

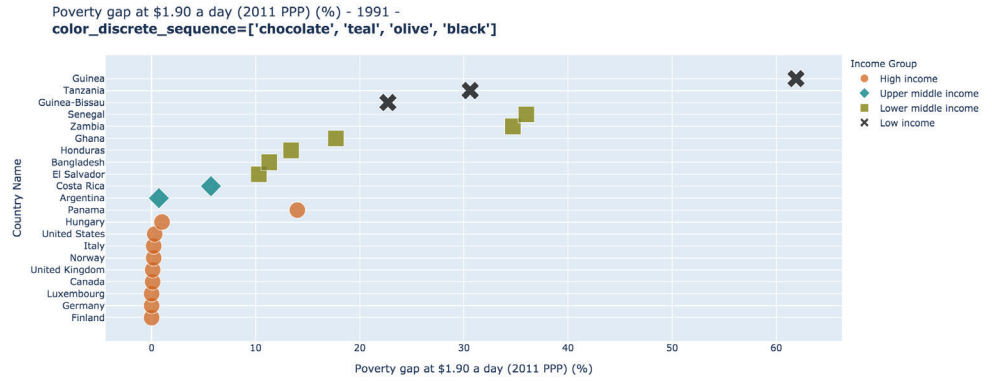


Рис. 6.15. Вручную установленные цвета для категориальных переменных

Выбирая цвета для графиков вручную, необходимо убедиться в том, что в списке будет столько же цветов, сколько уникальных значений переменной, которую вы визуализируете. В противном случае цвета начнут выбираться по кругу, что приведет к неразберихе. Обычно рекомендуется задавать предустановленные последовательности цветов для переменных, но ничто не мешает

вам создавать и собственные списки. До сих пор при установке цветов мы никак не влияли на то, каким цветом будет отображаться конкретный элемент в группе. Мы просто задавали последовательность цветов, а выбор происходил автоматически без вашего участия. Но иногда вам необходимо присвоить определенной категории конкретный цвет. Если вы знаете точные значения, то можете поставить им в соответствие определенные цвета, воспользовавшись параметром `color_discrete_map`, принимающим словарь, как показано ниже:

```
color_discrete_map={'High income': 'darkred',
                    'Upper middle income': 'steelblue',
                    'Lower middle income': 'orange',
                    'Low income': 'darkblue'}
```

Запуск кода с этой опцией приведет к результату, показанному на рис. 6.16.

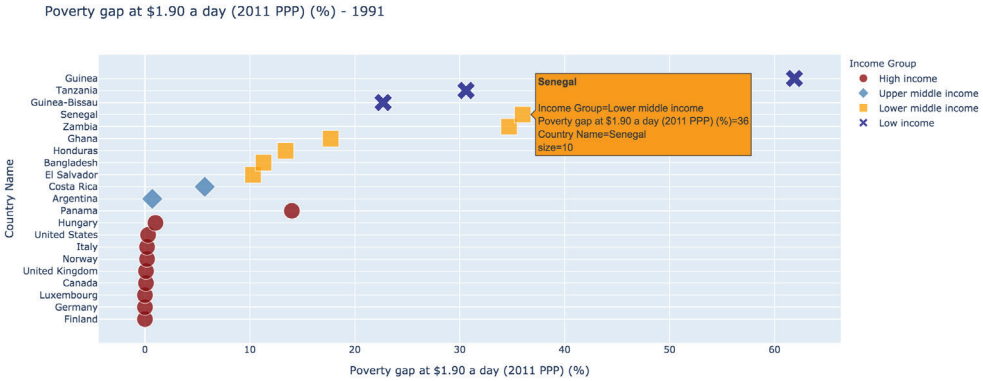


Рис. 6.16. Выбор конкретных цветов для указанных категориальных значений

Заметьте, что большинство параметров для работы с непрерывными переменными содержат слово `scale`, тогда как для дискретных используется слово `sequence`. Это поможет вам определиться с выбором правильных параметров в процессе установки цветовых соответствий.

В случае с непрерывными переменными пользователь может определить приблизительное значение точки по ее цвету, а также ее относительное положение в наборе данных. Например, он может быстро понять, что численность населения какой-то страны находится приблизительно на отметке в 20 млн человек и это одна из самых густонаселенных стран в наборе данных. Конечно, в любой момент пользователь может навести мышью на точку данных и узнать точное значение. В случае с дискретными данными мы заинтересованы в их разбиении на группы по этим переменным и отображении тенденций в рамках этих групп.

В этом разделе мы показали лишь малую часть того, что можно сделать на графиках с применением цветов. Теперь рассмотрим характерные для точечных диаграмм проблемы, связанные с наличием выбросов и количеством точек для отображения.

Управление наложениями и выбросами при помощи прозрачности, символов и масштаба

Скажем, нам необходимо проследить взаимосвязь между выбранной переменной и численностью населения за конкретный год. Для этого достаточно вынести переменную `Population, total` на ось *x*, а `perc_pov_19` – на ось *y*.

Но для начала создадим поднабор данных на основе датафрейма `poverty`, в котором выберем только 2010 год, а на флаг `is_country` наложим фильтр в виде значения `True`. Отсортируем полученные данные по численности населения:

```
df = poverty[poverty['year'].eq(2010) & poverty['is_country']].
sort_values('Population, total')
```

Посмотрим, какая зависимость наблюдается между интересующими нас переменными:

```
px.scatter(df,
           y=perc_pov_19,
           x='Population, total',
           title=' - '.join([perc_pov_19, '2010']),
           height=500)
```

Полученный в результате график показан на рис. 6.17.

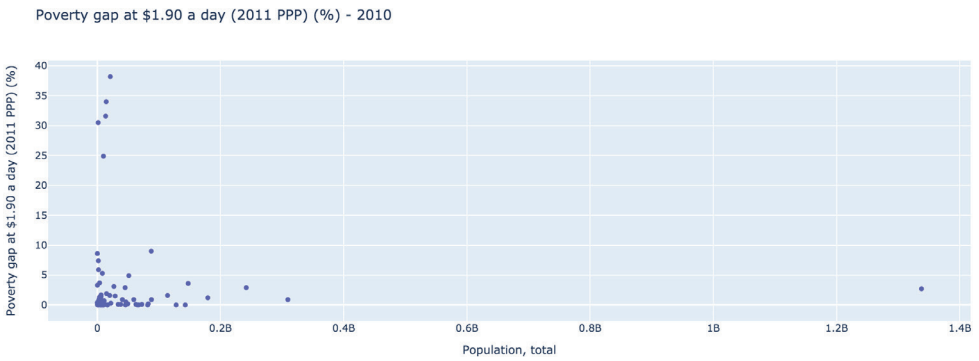


Рис. 6.17. Высокая плотность точек на графике и наличие выбросов

Присутствие единственного выброса по численности населения (Китай, 1,4 млрд человек) привело к скоплению остальных точек данных на ограниченном пространстве в левой части графика. Также легко заметить наличие небольшого кластера данных выше значения 25 по оси *y*, но в этом случае разброс не так критичен, как в случае с осью *x*. Другой проблемой является наложение большого количества маркеров друг на друга. Если отрисовывать маркеры непрозрачными цветами, они будут полностью накладываться друг на

друга, даже если их целая тысяча. Наличие этих двух проблем очень затрудняет чтение диаграммы.

В данном разделе мы рассмотрим разные техники обхода этих неудобств и расскажем о том, какие из них и в каких ситуациях стоит применять.

Как мы уже увидели, на нашем графике чрезмерно большое количество точек располагаются на ограниченном пространстве слева внизу, а значит, велика вероятность того, что некоторые из них полностью или частично накладываются друг на друга. Давайте посмотрим, что можно с этим сделать при помощи настройки прозрачности и размера маркеров.

Прозрачность и размер маркеров

Параметр `opacity`, отвечающий за *прозрачность маркеров*, принимает значения в интервале [0, 1] включительно. При этом значение 0 означает полную прозрачность или невидимость маркера либо группы маркеров, а 1 – полную непрозрачность, что эквивалентно цвету, присвоенному маркеру или группе. Что касается промежуточных значений, представляющих наибольший интерес, то установка параметру `opacity` значения 0,1 будет означать 10-процентную непрозрачность маркера. На практике это означает, что понадобится ровно 10 маркеров, наложенных друг на друга, чтобы восстановить исходный цвет заполнения. При установке значения 0,5 для достижения той же цели хватит двух маркеров.

Поскольку круги, характеризующие точки данных на нашем графике, очень маленькие, а значений на диаграмме выведено не так много, мы можем увеличить размер маркеров для лучшего визуального восприятия. Параметр `size`, как и все остальные параметры, может принимать имя столбца датафрейма или список числовых значений. Это еще один визуальный атрибут, который можно использовать для выражения значений в определенной колонке, при этом размер маркера будет соответствовать относительному значению точки данных. Часто такую диаграмму называют *пузырьковой* (bubble chart). В данном случае мы зададим фиксированный размер маркеров. Это можно легко сделать, передав параметру `size` список констант длиной, равной количеству строк в анализируемом датафрейме. В результате все маркеры станут одного размера, но он может не соответствовать нашим ожиданиям. Контролировать размер маркеров можно при помощи параметра `size_max`. Добавьте в вызов функции следующие строки:

```
opacity=0.1,
size=[5]*len(df),
size_max=15
```

В результате график приобретет вид, показанный на рис. 6.18.

Теперь диаграмма выглядит получше. Маркеры стали больше размером, а прозрачность 0,1 позволяет лучше определять наличие большого скопления точек данных – в нашем случае оно наблюдается возле начала координат. Возможно, есть и другие важные отличия в наших данных, но мы не можем их увидеть из-за наличия выбросов.

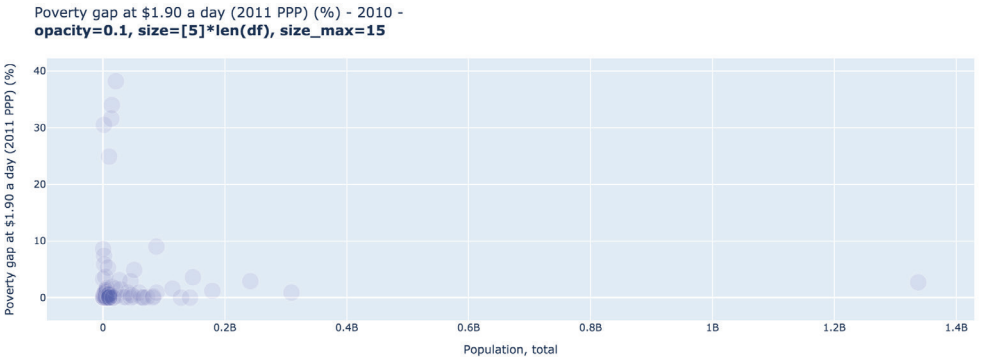


Рис. 6.18. Изменение прозрачности и размера маркеров

Вам всегда придется искать компромисс между прозрачностью маркеров и их заметностью. Чем более прозрачными будут ваши маркеры, тем лучше вы сможете видеть их большие скопления. В то же время они могут стать совсем прозрачными, и вы просто перестанете их видеть. С прозрачностью, установленной в 0,1, мы вплотную приблизились к этому порогу.

Теперь давайте взглянем на другую технику, включающую логарифмирование шкал на осях графика.

Использование логарифмических шкал

Традиционные шкалы на осях привычны всем и интуитивно понятны. Здесь все как в обычной жизни: если одна деталь вдвое длиннее другой, то она при прочих равных размерностях состоит из вдвое большего количества использованного материала. На предыдущих двух рисунках, к примеру, расстояние между 0 и 0,2 млрд было таким же, как между 0,2 млрд и 0,4 млрд. На нормальной шкале каждая очередная метка свидетельствует об увеличении показателя на фиксированную величину (в нашем случае это 0,2 млрд). Однако на *логарифмической шкале* (logarithmic scale) каждая следующая метка получается путем умножения значения предыдущей на константу.

Например, числа 10, 20, 30 и 40 формируют типичную последовательность, которую вы можете видеть на линейной шкале. Если бы шкала была логарифмической, то мы бы каждый раз не прибавляли 10, а умножали на 10. Таким образом, получили бы последовательность чисел 10, 100, 1000 и 10 000. Запуск кода из предыдущего примера с указанием параметра `log_x=True` привел бы к формированию графика, показанного на рис. 6.19.

Теперь наша диаграмма выглядит совсем иначе, несмотря на то что данные используются те же самые. Обратите внимание, что мы изменили значение параметра opacity на 0,25, поскольку значение 0,1 делает маркеры менее заметными. А в обновленном виде графика точки данных располагаются более распределенно, что и объясняет наш выбор. Мы видим, что больше всего маркеров скопилось в области 10 млн. В сравнении с 1,4 млрд это по-прежнему ничто, но на этом графике это не так бросается в глаза.

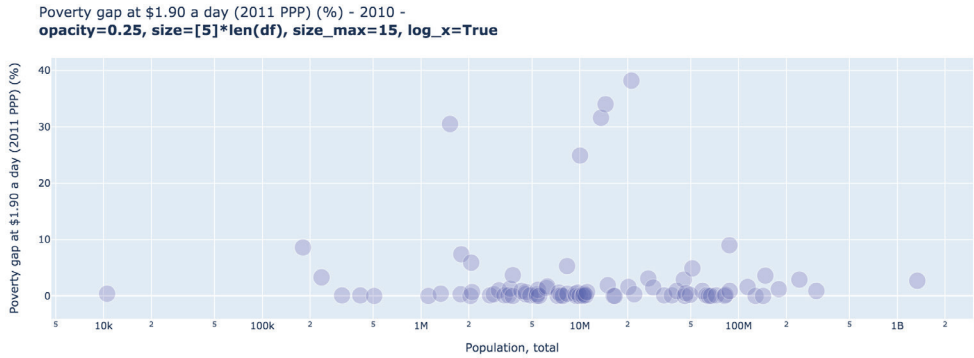


Рис. 6.19. Использование логарифмической шкалы

Обратите внимание, что значение каждой большой метки превосходит предыдущее в 10 раз (**10k**, **100k**, **1M**, **10M**, **100M** и **1B**). Грубо говоря, мы просто на каждой отметке добавляем один ноль. Также мы видим малые метки между ними с числами **2** и **5**, в которых находятся значения в два и пять раз больше, чем в предшествующей большой метке.

Теперь давайте рассмотрим еще одну возможность, которой можно воспользоваться в подобных ситуациях. На этот раз мы не будем использовать прозрачность, а освободим больше места при помощи указания фигуры или символа, который будет использоваться для маркеров. Это работает примерно так же, как с установкой цвета для дискретных переменных. Отвечает за символы маркеров параметр `symbol_sequence`, и в нем также осуществляется перебор переданных значений и сопоставление их каждому очередному значению из столбца. Мы передадим ему список, состоящий из одного элемента, так что для всех маркеров будет выбран один и тот же символ окружности.

Итак, удалите из списка вызова функции параметр `opacity` и замените его на `symbol_sequence= ['circle_open']`. В результате получится график, показанный на рис. 6.20.

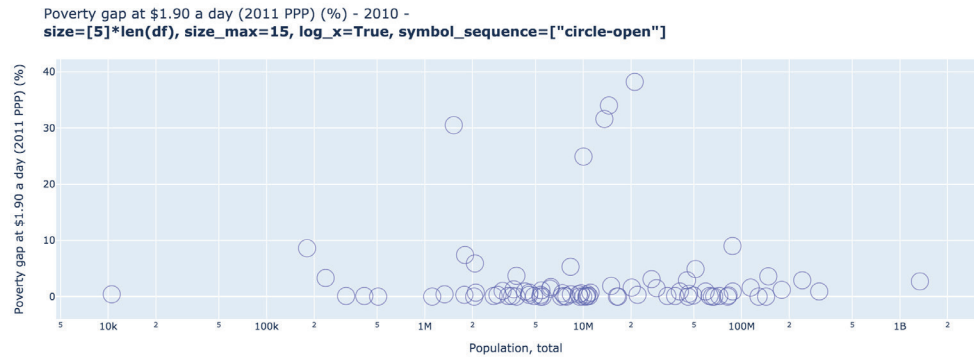


Рис. 6.20. Изменение символа для маркеров

Так вышло даже лучше, поскольку нам больше не приходится жертвовать видимостью маркеров, делая их прозрачными. Кроме того, мы достигли своей

цели, так как теперь хорошо видим места наибольшего скопления точек данных. Логарифмическая шкала на оси `x` помогла более равномерно распределить маркеры по горизонтали, что также облегчает чтение графика. Подписи под метками понятно указывают на значения, но нам может потребоваться дать пользователям, незнакомым с логарифмическими шкалами, какие-то дополнительные подсказки.

Можно даже предоставить пользователю возможность самому настроить все опции, о которых мы упомянули выше: добавить компоненты, с помощью которых он сможет настроить прозрачность маркеров, используемые символы, переключаться между линейной и логарифмической шкалой и т. д. Но нужно сначала проработать графики самому и предоставить подходящие значения по умолчанию для этих управляющих компонентов.

Мы знаем, что наш график выводит информацию о странах и что их не может быть больше 200. Так что мы можем заранее настроить прозрачность маркеров под величины такого порядка. Если бы у нас были тысячи точек данных, параметр `opacity`, вероятно, пришлось бы снизить где-то до 0,02. К тому же окружности очень неплохо сработали в нашем примере, так что мы могли бы использовать этот символ по умолчанию и вовсе забыть о прозрачности. То же самое можно сказать и о параметре `size`. Зная, что мы оперируем данными о численности населения, а значит, у нас всегда будут какие-то выбросы, можно использовать логарифмическую шкалу в качестве опции по умолчанию.

Более того, мы можем предоставить пользователю возможность выбирать индикаторы, которые он хочет анализировать. Однако стоит учитывать, что в этом случае ему придется более детально вдаваться в подробности.

Итак, мы довольно много сделали, чтобы проанализировать выбранные показатели для стран, и вы можете использовать все полученные знания при создании интерактивных дашбордов. Теперь пришло время поговорить еще об одном интерактивном компоненте – слайдере.

Знакомство со слайдерами, включая слайдеры диапазонов

Компоненты `Slider` и `RangeSlider` по своей сути представляют ползунок, который пользователь может перемещать вдоль горизонтального или вертикального *слайдера* с целью изменить его текущее значение. Обычно эти элементы управления используются для установки непрерывных значений, поскольку их внешний вид и способ взаимодействия отлично для этого подходят. Но это не строгое правило, и слайдеры можно прекрасно использовать и для выбора категориальных/дискретных значений. Как вы помните из предыдущего примера, у нас есть три индикатора, начинающихся с `perc_pov_`, а также мы знаем, за какие годы в нашем датафрейме собраны данные. Сейчас мы создадим два слайдера, первый из которых будет предназначаться для выбора индикатора, а второй – для выбора года. Комбинация из индикатора и года будет использоваться для фильтрации исходного датафрейма и обновления графика. На рис. 6.21 показана верхняя часть секции, над которой мы сейчас будем работать.

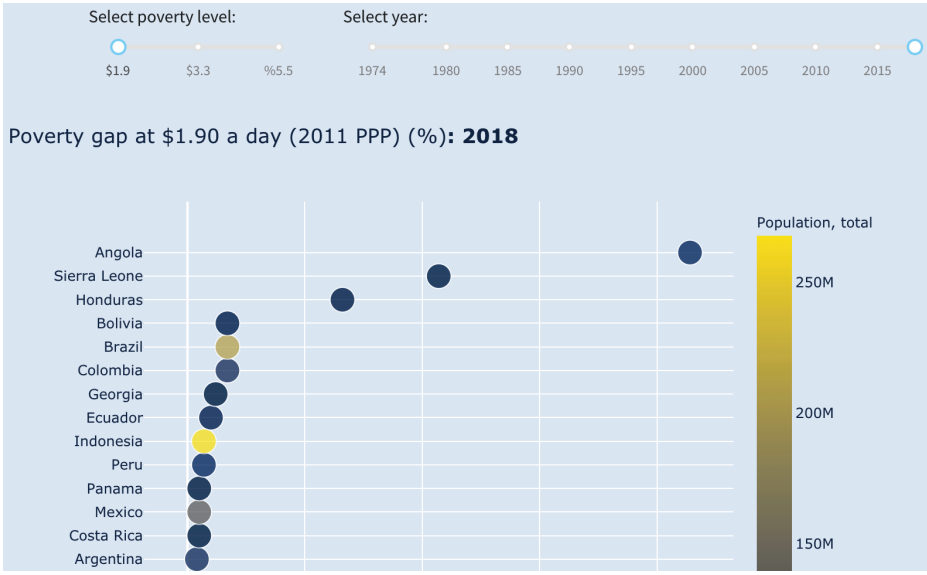


Рис. 6.21. Два слайдера для управления графиком

Как видите, запланированный функционал требует создания трех основных компонентов: двух слайдеров и одного графика. Конечно, у нас будут и другие компоненты, включая метки, но главным образом мы сосредоточимся на этих интерактивных составляющих.

> **Важно**
>
> Компонент `RangeSlider` является почти полным аналогом компонента `Slider`. Основное отличие между ними состоит в том, что у `RangeSlider` может быть более одного управляющего ползунка для возможности указания целого диапазона значений. Сейчас мы будем довольствоваться компонентом `Slider`, а о его разновидности `RangeSlider` поговорим в следующих главах.

Как и всегда, начнем разрабатывать задуманный функционал в JupyterLab, а по готовности перенесем его в наше приложение. Но сначала познакомимся с компонентом `Slider` и создадим макет приложения.

Вы можете создать минимальное приложение с единственным компонентом `Slider`, как мы уже делали ранее с другими компонентами. Для этого просто воспользуйтесь вызовом `dcc.Slider()`, как показано ниже:

```
app = JupyterDash(__name__)
app.layout = html.Div([
    dcc.Slider()
])
app.run_server(mode='inline')
```

Это позволит создать приложение с одним слайдером, что видно на рис. 6.22.

Рис. 6.22. Базовый компонент `Slider`

Это один из самых интуитивно понятных элементов управления – хватай за ползунок и перемещай его по оси. Но нам не хватает дополнительной информации о том, какими значениями ограничивается выбор. Давайте это исправим. Начнем с создания первого нашего слайдера для выбора индикатора. Взглянем на параметры, которые будем использовать при его создании:

- `min`: минимальное значение слайдера;
- `max`: максимальное значение слайдера;
- `step`: отвечает за шаг, с которым будет происходить смена значений на пути от `min` к `max`. По умолчанию шаг установлен в единицу, но вы можете сделать его больше или меньше. К примеру, если вы хотите создать слайдер для контроля за прозрачностью маркеров, можно задать следующие параметры: `min=0`, `max=1` и `step=0.01`. Это позволит пользователю выбрать один из ста вариантов;
- `dots`: этот параметр указывает на то, будут ли на слайдере метки или он будет представлять собой просто линию. В нашем случае пользователь должен сделать выбор из трех значений, так что метки лучше оставить;
- `included`: обратите внимание, что на слайдере, показанном на рис. 6.22, левая часть от ползунка окрашена в голубой цвет, а правая – в серый. При перемещении ползунка голубая область следует за ним, и такое поведение заложено по умолчанию. В нашем случае у пользователя будет выбор из трех элементов, так что закрашивать область слева от ползунка нам ни к чему. Поэтому дадим этому параметру значение `False`;
- `value`: при помощи этого параметра можно установить значение, которое будет выбрано на слайдере по умолчанию.

Ниже показан код для слайдера с выбором значения из диапазона от 0 до 10:

```
dcc.Slider(min=0,
           max=10,
           step=1,
           dots=True,
           included=False)
```

В результате будет создан слайдер, показанный на рис. 6.23.

Рис. 6.23. Слайдер с пользовательскими опциями

Точки на слайдере подсказывают пользователю, что он выбирает из ограниченного числа вариантов, а отключенный параметр `included` помогает не отвлекать его внимание на закрашенную область слева от ползунка.

Еще одним важным параметром слайдера является `marks`, отвечающий за подписи к меткам. Иногда в приложении не находится достаточно места для вывода всех подписей. В таких случаях можно выводить только некоторые. Мы продемонстрируем это на примере слайдера с годами, но сначала давайте создадим слайдер для выбора индикаторов. Сперва сделаем это без параметра `marks`, его добавим позже:

```
dcc.Slider(id='perc_pov_indicator_slider',
           min=0,
           max=2,
           step=1,
           value=0,
           included=False)
```

При выборе идентификатора (`id`) слайдера мы последовали нашему правилу и начали его с `perc_pov_`, чтобы не нарушалась общая система именования элементов в приложении. Значения, которые функция обратного вызова будет принимать от этого компонента, находятся в интервале от 0 до 2, что мы задали при помощи параметров `min`, `max` и `step`. Сами эти значения ничего не значат, нам ведь нужны полные названия индикаторов. А получить их можно из ранее созданного списка `perc_pov_cols` по индексу, в качестве которого будет выступать идентификатор выбранного в слайдере элемента. Посмотрим, как это работает, позже, когда будем писать для нашего слайдера функцию обратного вызова. А пока займемся созданием подписей к меткам.

Настройка подписей и значений слайдеров

Самый простой способ подписать метки на слайдере – создать словарь следующего вида: `{0: '$1.9', 1: '$3.2', 2: '$5.5'}`. Ключи словаря будут соответствовать значениям атрибута `value`, а значения будут выводиться в качестве подписей пользователю. Для нашего случая это подходит.

Также вы можете при желании управлять стилем отображения подписей с помощью атрибута `style`, принимающего CSS-инструкцию в виде словаря. Если вы посмотрите на рис. 6.21, то увидите, что подписи к меткам обоих слайдеров написаны довольно светлым цветом, и даже создается ощущение, что они принадлежат одному слайдеру. Мы можем это изменить, сделав цвет подписей более темным. Кроме того, для слайдера с индикаторами мы можем установить полужирный шрифт подписей. Это в том числе поможет визуально отделить слайдеры друг от друга и подчеркнуть их индивидуальность. С выбором года пользователь, скорее всего, быстро разберется, а вот об индикаторах уровня бедности он может ничего не знать.

Нам нужно выбрать цвет, который не будет идти вразрез с нашим графиком. И поскольку мы используем цветовую шкалу **cividis**, то можем взять цвет из нее и заодно показать, как это делается. В модуле `px.colors.sequential` помимо прочего содержатся списки цветов в последовательных цветовых шкалах. Рассмотреть состав цветовой схемы **cividis** можно при помощи следующей инструкции:

```
px.colors.sequential.Cividis
['#00224e',
 '#123570',
 '#3b496c',
 '#575d6d',
 '#707173',
 '#8a8678',
 '#a59c74',
 '#c3b369',
 '#e1cc55',
 '#fee838']
```

Этот список содержит десять цветов, составляющих эту цветовую шкалу. Если помните, мы создавали подобные списки из двух-трех цветов. Также важно знать, что вы можете воспользоваться и обратной последовательностью цветов из шкалы, добавив к ее имени `_r`. Например, вы можете использовать шкалу `px.colors.sequential.Cividis_r`. Сам градиентный переход при этом останется прежним, но желтые цвета будут соответствовать нижней части спектра.

Давайте используем для подписей меток на слайдере самый темный цвет этой цветовой палитры. Присвоим его переменной, как показано ниже:

```
cividis0 = px.colors.sequential.Cividis[0]
```

Теперь можно передать значение параметру `marks` следующим образом:

```
marks={0: {'label': '$1.9', 'style': {'color': cividis0, 'fontWeight': 'bold'}},
       1: {'label': '$3.2', 'style': {'color': cividis0, 'fontWeight': 'bold'}},
       2: {'label': '$5.5', 'style': {'color': cividis0, 'fontWeight': 'bold'}}}
```

По сути, мы просто расширили словарь, вставив вместо текста подписей словари следующего вида:

```
{'label': <label>, 'style': {<attribute_1>: <value_1>, <attribute_2>: <value_2>}
```

> **Важно**
>
> Обычно атрибуты CSS, такие как `font-size` и `font-weight`, отделяются дефисом и записываются строчными буквами. В Dash вы можете использовать те же атрибуты, но с использованием «верблюжьего стиля» (`fontSize` и `fontWeight`), что видно в примере выше.

Теперь давайте создадим второй слайдер с похожими настройками. Для начала объявим датафрейм, содержащий эти переменные:

```
perc_pov_df = poverty[poverty['is_country']].dropna(subset=perc_pov_cols)
perc_pov_years = sorted(set(perc_pov_df['year']))
```

Важно, что мы избавились от пропущенных значений и создали отсортированный список уникальных годов `perc_pov_years`, воспользовавшись для этого функциями `sorted` и `set`.

Показанный ниже код служит для создания слайдера по годам:

```
dcc.Slider(id='perc_pov_year_slider',
           min=perc_pov_years[0],
           max=perc_pov_years[-1],
           step=1,
           included=False,
           value=2018,
           marks={year: {'label': str(year),
                         'style': {'color': cividis0}}
                  for year in perc_pov_years[::5]})
```

Здесь все почти так же, как со слайдером для индикаторов. По умолчанию мы установили 2018 год, поскольку он последний в нашем наборе данных. Если бы наше приложение обновлялось динамически, можно было бы установить по умолчанию максимальное значение из списка `perc_pov_years`. Обратите внимание, что мы воспользовались срезом для отображения каждого пятого года из списка. В противном случае нам было бы очень трудно использовать слайдер. На рис. 6.24 представлены обновленные элементы управления.

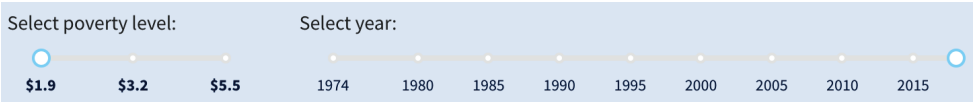


Рис. 6.24. Слайдеры с обновленными цветами

Осталось добавить компонент `Graph` следующим образом:

```
dcc.Graph(id='perc_pov_scatter_chart')
```

Как мы уже упоминали, в нашем приложении также присутствуют вспомогательные компоненты `Label`, `Col` и `Row`, но о них мы уже рассказывали ранее и не будем вновь подробно на них останавливаться.

Теперь мы готовы к написанию функции обратного вызова, которая объединит все три компонента, которые мы разместили на макете.

1. Для начала создадим декоратор функции. Здесь для вас не будет ничего нового. Небольшое отличие состоит в том, что в данном случае у нас будет сразу два элемента ввода. Порядок следования параметров при определении функции должен строго совпадать с порядком элементов `Input` в декораторе, так что назовем их соответствующе:

```
@app.callback(Output('perc_pov_scatter_chart', 'figure'),
              Input('perc_pov_year_slider', 'value'),
              Input('perc_pov_indicator_slider', 'value'))
```

2. Далее создадим сигнатуру функции и напишем ее первые строки. Входные параметры назовем `year` и `indicator`. В начале функции воспользуемся значением параметра `indicator` для извлечения названия индикатора из списка `perc_pov_cols`. После этого создадим поднабор данных `df`, в котором оставим из датафрейма `perc_pov_df` только строки с нужным нам годом без пропущенных значений и в отсортированном виде. В наших данных есть год, за который нет информации, но мы должны были включить его в слайдер. Таким образом, мы должны учесть то, что пользователь может его выбрать. Это сделано при помощи условия `if df.empty`, как показано ниже:

```
def plot_perc_pov_chart(year, indicator):
    indicator = perc_pov_cols[indicator]
    df = (perc_pov_df
         [perc_pov_df['year'].eq(year)]
         .dropna(subset=[indicator])
         .sort_values(indicator))
    if df.empty:
        raise PreventUpdate
```

3. Теперь, когда у нас есть готовый датафрейм, можно создать объект `Figure` и вернуть его. Большая часть кода из следующего фрагмента будет вам понятна. Параметр `hover_name` используется для добавления заголовка всплывающему окну, которое будет показываться при наведении мышью на точки данных. Задав для этого параметра значение `'Country Name'`, мы выведем в заголовке окна название соответствующей страны жирным шрифтом. Для динамического указания высоты диаграммы мы воспользовались тем же трюком, что и в предыдущей главе, а именно задали фиксированную высоту и добавили по 20 пикселей на каждую страну. Имя параметра `ticksuffix` говорит само за себя – здесь мы можем задать окончание для каждого значения на оси *x*. Укажем знак процента:

```
fig = px.scatter(df,
                 x=indicator,
                 y='Country Name',
                 color='Population, total',
                 size=[30]*len(df),
                 size_max=15,
                 hover_name='Country Name',
                 height=250 +(20*len(df)),
                 color_continuous_scale='cividis',
                 title=indicator + '<b>: ' + f'{year}' +'</b>')
```

```
    fig.layout.paper_bgcolor = '#E5ECF6'
    fig.layout.xaxis.ticksuffix = '%'
    return fig
```

Добавив элементы интерфейса на макет и функцию обратного вызова, мы тем самым создали новую секцию в нашем приложении – с двумя интерактивными слайдерами, с помощью которых в сумме можно построить более 130 различных графиков. На рис. 6.25 представлен финальный результат.

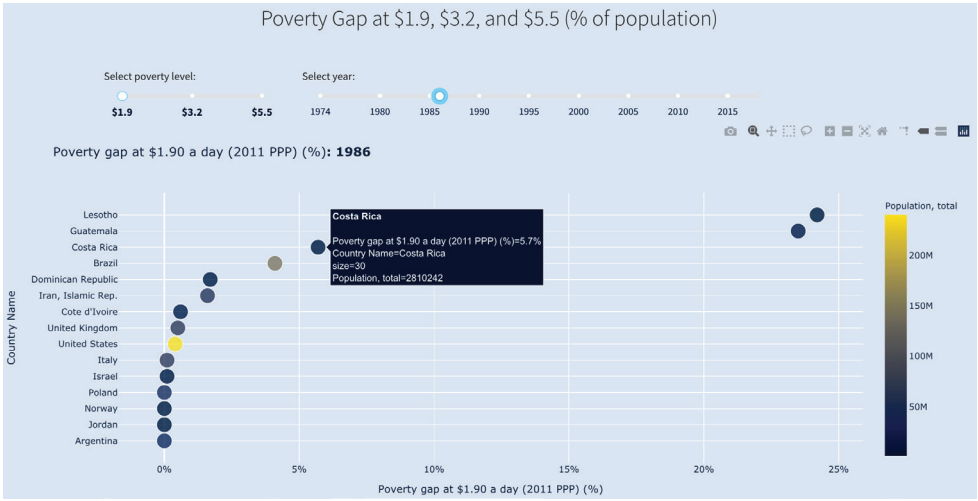


Рис. 6.25. Два слайдера и точечная диаграмма

Поздравляем с еще одним пополнением в вашем приложении! Мы создали нашу первую функцию обратного вызова с несколькими элементами ввода, чем значительно обогатили возможности пользователя без излишнего усложнения интерфейса приложения.

В плане функционала между выпадающими списками и слайдерами нет фактически никаких отличий. Все то же самое мы могли реализовать и с помощью списков, и все работало бы прекрасно. Главным преимуществом выпадающих списков является их способность экономить место на макете. Небольшой прямоугольник может содержать в себе десятки и сотни вариантов для выбора. Кроме того, они могут иметь достаточно длинные названия, которые невозможно было бы уместить в виде подписей рядом на слайдере.

С другой стороны, слайдеры представляют хорошо понятную информацию. Они в неявной форме содержат метаданные об имеющихся у пользователя возможностях. Взглянув на слайдер, вы сразу видите, в каких пределах можно менять значение и насколько велик этот интервал. При выборе значения вы легко можете сказать, как оно соотносится с другими доступными вариантами. К примеру, в случае с уровнями бедности перед вашими глазами находятся сразу все возможные опции.

И наконец, управление слайдерами очень приближено к тому, как мы взаимодействуем с физическими объектами, а значит, у пользователя не будет возникать никаких вопросов относительно работы с этими элементами интер-

фейса. В общем и целом в процессе выбора компонента вы должны руководствоваться имеющимся местом на макете, типом переменных, которые хотите визуализировать, и тем, как именно они должны быть отображены.

На этот раз мы не будем рассказывать о том, как перенести все произведенные изменения в ваше существующее приложение. Мы сделали это намеренно. Ранее мы уже неоднократно выполняли эти действия, и теперь вы без труда справитесь с этим самостоятельно. Если у вас возникнут вопросы, вы всегда сможете обратиться за ответами в наш репозиторий.

Давайте подытожим то, что вы узнали при чтении этой главы.

Заключение

В данной главе мы познакомились с точечными диаграммами или диаграммами рассеяния, узнали, как их создавать при помощи модуля `graph_objects` и посредством Plotly Express. Также мы рассмотрели способы создания графиков с присутствием нескольких рядов данных. После этого мы посвятили немало времени настройке цветового оформления диаграмм и узнали о различиях, характерных для визуализации категориальных (дискретных) и количественных (непрерывных) переменных. Мы рассмотрели разные типы шкал: последовательную, расходящуюся и категориальную. Попутно мы узнали, как можно устанавливать свои собственные цвета, последовательности и шкалы. Вместе с тем обсудили и успешно решили ряд проблем, связанных с наличием выбросов и чрезмерным наложением данных на графиках. В процессе этого мы поэкспериментировали с прозрачностью маркеров, их символами и размером, а также посмотрели, каких сложностей позволяет избежать логарифмическая шкала.

После этого мы познакомились с таким полезным элементом интерфейса, как слайдер, и создали отдельную секцию в нашем приложении с двумя слайдерами и графиком, полностью зависящим от выбора пользователя в этих слайдерах. Это было реализовано при помощи функции обратного вызова, оперирующей сразу двумя элементами ввода.

Освоив все эти техники, вы сможете создавать полноценные интерактивные дашборды, которые частично или полностью способны заменить слайды и презентации. Научившись настраивать внешний вид макетов, вы можете размещать элементы в интерфейсе по своему усмотрению и подбирать для них наиболее подходящие размеры.

Все типы диаграмм, которые мы изучали до сих пор, использовали традиционные геометрические фигуры, такие как окружности, линии и прямоугольники. В следующей главе мы обратимся к нетрадиционным фигурам и научимся визуализировать данные на картах. Географические карты являются довольно узнаваемым инструментом визуализации данных, но работать с ними немного сложнее по сравнению с привычными всем геометрическими фигурами. Тем интереснее…

Глава 7

Работа с географическими картами и обогащение дашбордов при помощи языка разметки Markdown

В этой главе мы посвятим свое время географическим картам как одному из наиболее привлекательных типов визуализации. Существует множество способов представления данных на картах и самих типов карт. Есть масса специализированных географических и научных приложений для работы с картами. Мы же главным образом сосредоточимся на двух типах карт: *картограмме* (choropleth map plot) и *точечной карте* (scatter map plot). С картограммами мы все хорошо знакомы. Это такие карты, на которых географические области закрашиваются в определенные цвета для выделения стран, регионов, районов или других произвольных полигонов с целью демонстрации какой-то количественной характеристики. Большая часть полученных в предыдущей главе книги знаний может быть использована вами при создании точечных карт, поскольку они отвечают тем же принципам, хоть и с небольшими отличиями. Вместо осей *x* и *y* здесь у нас есть широта и долгота, к тому же мы можем использовать различные проекции карт.

Также в этой главе мы познакомимся с новым компонентом из библиотеки Dash Core Component под названием **Markdown**.

Кроме того, мы узнаем, как пользоваться библиотекой **Mapbox**, предлагающей богатейшие возможности со слоями, темами и уровнями приближения карт. С помощью этой библиотеки также очень просто можно строить картограммы и точечные карты.

Давайте начнем, но сначала, как обычно, перечислим темы, которые будут рассмотрены в главе:

- знакомство с картограммами;
- использование анимации для добавления нового слоя в визуализацию;
- использование функций обратного вызова с картами;
- создание компонента Markdown;
- знакомство с проекциями карты;

- использование точечных карт;
- использование карт Mapbox;
- другие опции и инструменты для работы с картами;
- внедрение интерактивной карты в приложение.

Технические требования

В данной главе мы продолжим использовать инструменты и пакеты, с которыми познакомились в предыдущих главах. При создании визуализаций мы по большей части будем использовать Plotly Express. Пакеты, которые мы также будем применять: Plotly, Dash, Dash Core Components, Dash HTML Components, Dash Bootstrap Components, pandas и JupyterLab.

Исходный код к этой главе располагается в хранилище GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_07**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3sAY8z8**.

Давайте начнем с построения картограммы по странам.

Знакомство с картограммами

Картограмма (choropleth map) представляет собой карту с закрашенными полигонами, обозначающими конкретные области. Plotly поставляется с картой стран, включая штаты США, так что с наличием информации о странах построить первую картограмму не составит труда. В нашем датасете вся необходимая информация имеется. В каждой строке у нас есть название и код страны, а также год, метаданные о странах, включая регион, уровень доходов и т. д., и данные по индикаторам. Иными словами, каждая точка данных у нас тесно связана с географическим местоположением. Начнем с выбора года и индикатора и посмотрим, как данные по этому индикатору меняются в зависимости от страны.

1. Создадим датафрейм на основе набора данных `poverty` и объявим переменные `year` и `indicator`:

```
import pandas as pd
poverty = pd.read_csv('data/poverty.csv')
year = 2016
indicator = 'GINI index (World Bank estimate)'
```

2. Ограничим датафрейм данными по выбранному году и только по странам:

```
df = poverty[poverty['is_country'] & poverty['year'].eq(year)]
```

3. Создадим картограмму при помощи функции `choropleth` из состава Plotly Express, указав столбец, отвечающий за страны, и столбец, который будет использоваться для цветовой дифференциации:

```
import plotly.express as px
px.choropleth(df, locations="Country Code", color=indicator)
```

На рис. 7.1 представлена визуализация, которая будет создана в результате таких простых действий.

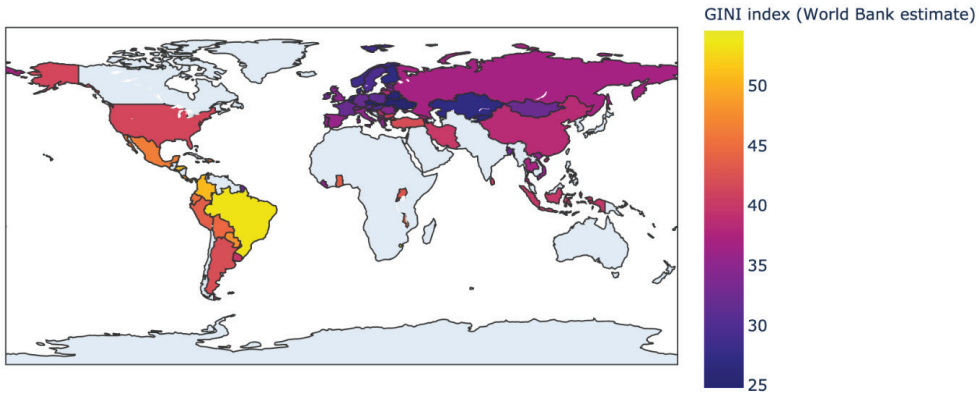


Рис. 7.1. Картограмма с данными по странам

Коды стран, которые мы использовали, есть в Plotly, и они имеют трехбуквенный формат ISO. Как и в случае с точечными диаграммами, вы можете видеть, что передача столбца с числовыми значениями для разделения по цветам привела к выбору непрерывной цветовой шкалы. Если бы мы дали для цветов категориальные данные, была бы использована дискретная шкала. Вы можете это легко проверить – передайте функции параметр `color='Income Group'`, и вы получите график, показанный на рис. 7.2.

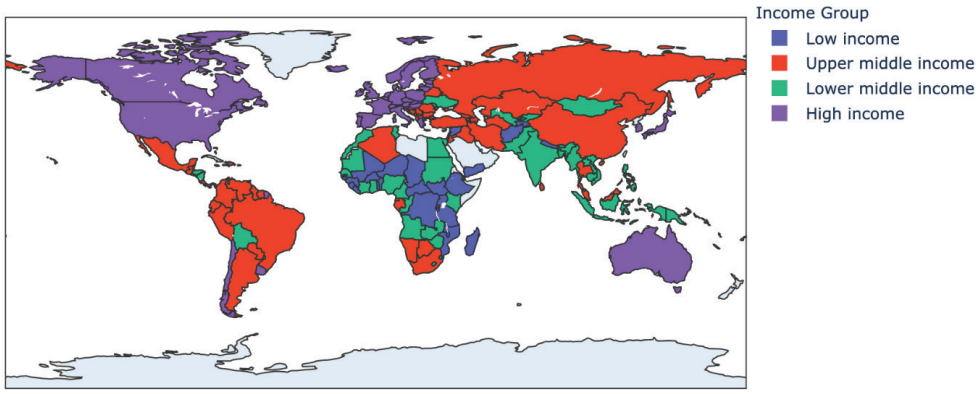


Рис. 7.2. Картограмма по странам с дискретной цветовой последовательностью

Как видите, система выбора цветов здесь работает так же точно, как и с точечными диаграммами.

Также мы можем использовать для отрисовки на карте названия стран, а не их коды. Для этого передайте функции параметр `locationmode='country names'`, а все остальное останется неизменным:

```
px.choropleth(df,
              locations=['Australia', 'Egypt', 'Chile'],
              color=[10, 20, 30],
              locationmode='country names')
```

В результате мы получим картограмму, показанную на рис. 7.3.

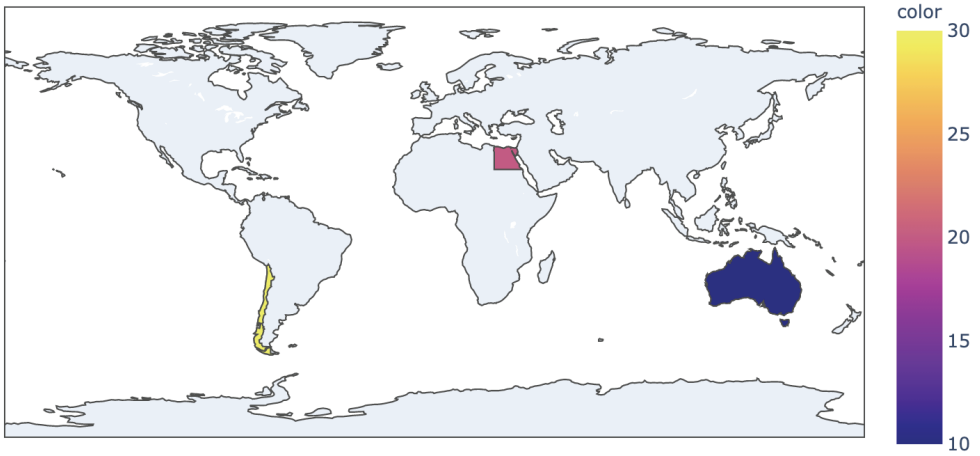


Рис. 7.3. Картограмма по странам с указанием их названий

Цветовая шкала в данном случае получила заголовок `color`, поскольку не понятно, что за характеристика имеется в виду, но это не имя столбца из датафрейма. Мы можем переименовать шкалу, указав параметр `labels={'color': <metric_name>}`, чтобы явно задать используемый показатель. Теперь давайте посмотрим, как можно добавить картам интерактивности без использования функций обратного вызова.

Использование анимации для добавления нового слоя в визуализацию

В предыдущих примерах мы фиксировали конкретный год в переменной и производили анализ показателя только по одному этому году. Но, как вы догадываетесь, поскольку в наших данных есть сведения о годах, мы можем использовать их в качестве группирующей переменной – для этого достаточно воспользоваться параметром `animation_frame`, который поможет добавить нашей карте интерактивности. В результате в нижней части карты появится анимационный слайдер, с помощью которого вы можете выбрать нужный вам год или нажать на кнопку проигрывания, тем самым запустив анимацию с перемещением во времени. По сути, вы увидите сменяющие друг друга слайды, как на видео или презентации. На самом деле для каждого года из последовательности собирается свой датафрейм и слайды по ним автоматически сменяют друг друга. Ниже представлен обновленный фрагмент кода:

```
px.choropleth(poverty[poverty['is_country']],
              color_continuous_scale='cividis',
              locations='Country Code',
              color=indicator,
              animation_frame='year')
```

На рис. 7.4 показан результат, который увидите на экране.

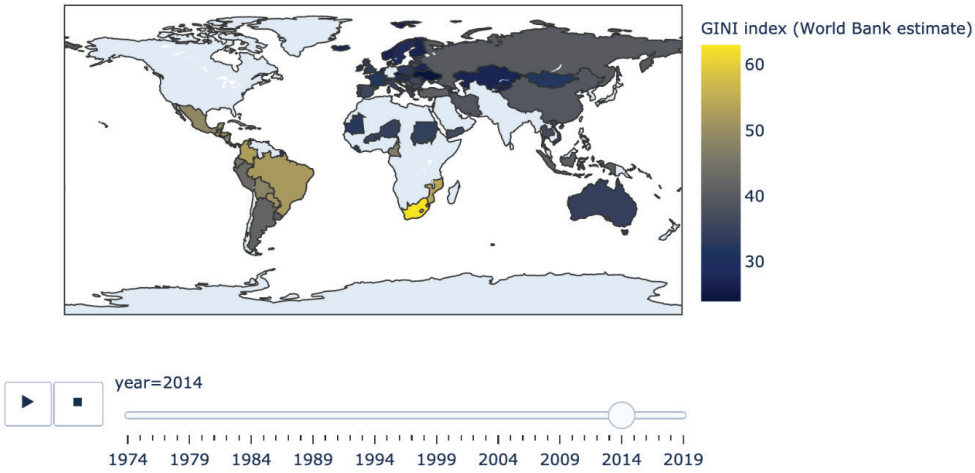


Рис. 7.4. Картограмма по странам с анимацией

Как видите, достаточно было указать столбец с данными для параметра `animation_frame`, а все остальное произошло само собой. При этом мы использовали датафрейм, в котором были данные по странам и всем годам. Дальнейшая фильтрация данных была выполнена уже по столбцу из параметра `animation_frame`. Заметьте, что мы также ради эксперимента сменили цветовую шкалу. Обе использованные шкалы должны быть хорошо различимы и в черно-белых источниках.

Теперь, когда мы построили базовую карту, давайте рассмотрим опции, которыми мы можем управлять. У атрибута `layout` есть вложенный атрибут `geo`, который открывает нам доступ ко всему разнообразию контроля над различными аспектами географических карт. Эти атрибуты работают так же, как и все остальные.

Установка нужных нам атрибутов выполняется по привычной схеме – для этого используется следующая конструкция: `fig.layout.geo.<attribute> = value`. Давайте зададим несколько атрибутов, после чего посмотрим, как изменится наша визуализация:

- удалим прямоугольную рамку вокруг карты:

```
fig.layout.geo.showframe = False
```

- отобразим границы стран, даже для тех из них, по которым у нас нет данных:

```
fig.layout.geo.showcountries = True
```

- используем другую проекцию земного шара, выбрав тип проекции `natural earth`. Подробнее о проекциях мы будем говорить позже:

```
fig.layout.geo.projection.type = 'natural earth'
```

- ограничим вертикальный диапазон на карте, чтобы лучше были видны страны. Для этого зададим минимальную и максимальную широту для отображения на карте:

```
fig.layout.geo.lataxis.range = [-53, 76]
```

- ограничим горизонтальный диапазон просмотра, воспользовавшись той же техникой:

```
fig.layout.geo.lonaxis.range = [-137, 168]
```

- изменим базовый цвет на белый, чтобы было отчетливо видно, по каким странам у нас нет данных:

```
fig.layout.geo.landcolor = 'white'
```

- установим цвет фона карты (цвет океана) и внешнего фона графика. Используем для них цвет фона приложения, чтобы все смотрелось последовательно:

```
fig.layout.geo.bgcolor = '#E5ECF6'
fig.layout.paper_bgcolor = '#E5ECF6'
```

- установим серый цвет для границ стран и береговой линии:

```
fig.layout.geo.countrycolor = 'gray'
fig.layout.geo.coastlinecolor = 'gray'
```

- поскольку заголовок цветовой шкалы получился довольно длинным, заменим в нем пробелы на символы переноса строки:

```
fig.layout.coloraxis.colorbar.title = indicator.replace(' ', '<br>')
```

В результате мы получим гораздо более опрятную визуализацию карты, показанную на рис. 7.5.

Для создания этой карты нам потребовалось написать всего несколько строк кода. Мы ограничили диапазоны просмотра, чтобы все страны были отчетливо видны. Также мы закрасили нужные нам области и выделили границы стран. Есть и другие полезные опции для настройки в атрибуте `fig.layout.geo`. Но мы перейдем к динамическому выбору индикатора для анализа.

Использование функций обратного вызова с картами

До сих пор мы работали на картах с одним выбранным индикатором из столбца датафрейма. В то же время мы можем предоставить выбор индикатора для

анализа пользователю. Переменную `year` мы уже сделали интерактивной при помощи параметра `animation_frame`. По сути, наша карта может стать первым по-настоящему исследовательским интерактивным графиком, с которым пользователь начнет работать в нашем приложении и сможет анализировать выбранные показатели и их динамику.

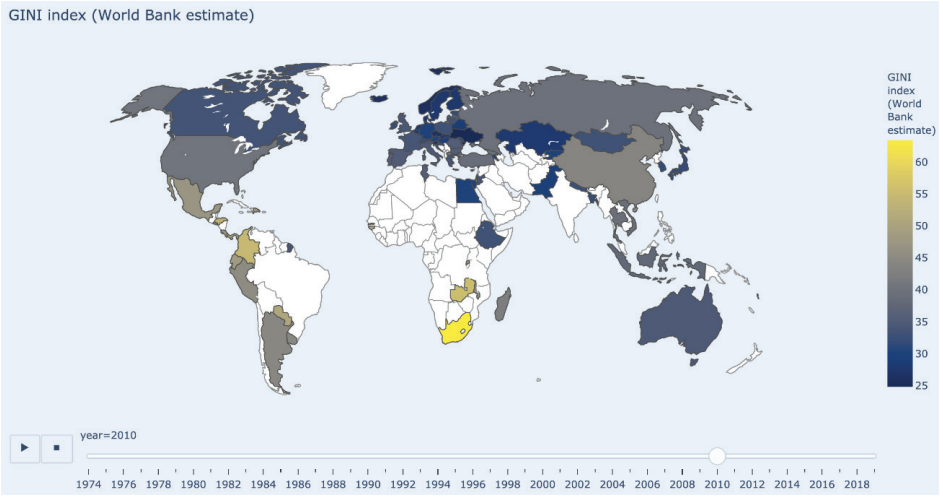


Рис. 7.5. Картограмма по странам с анимацией и дополнительными настройками

Сначала мы реализуем базовый интерактивный функционал, после чего посмотрим, как можно использовать компонент `Markdown` для добавления дополнительного контекста на карту.

Давайте внедрим задуманный функционал в нашем приложении в JupyterLab.

1. Создадим компонент `Dropdown`, который заполним названиями столбцов из датафрейма `poverty` с порядковыми номерами от 3 до 54:

```
dcc.Dropdown(id='indicator_dropdown',
             value='GINI index (World Bank estimate)',
             options=[{'label': indicator, 'value': indicator}
                      for indicator in poverty.columns[3:54]])
```

2. Теперь создадим компонент `Graph` чуть ниже:

```
dcc.Graph(id='indicator_map_chart')
```

3. Названия индикаторов могут быть достаточно длинными и занимать половину экрана. Мы можем справиться с этим так же, как делали ранее, создав для этого отдельную функцию. Эта функция будет принимать на вход строку, разбивать ее на слова, группировать их в тройки и ставить символы переноса строки после каждой группы:

```
def multiline_indicator(indicator):
    final = []
    split = indicator.split()
```

```
    for i in range(0, len(split), 3):
        final.append(' '.join(split[i:i+3]))
    return '<br>'.join(final)
```

4. Напишем декоратор функции обратного вызова, соединяющей выпадающий список с картой:

```
@app.callback(Output('indicator_map_chart', 'figure'),
              Input('indicator_dropdown', 'value'))
```

5. Напишем саму функцию, принимающую на вход выбранный индикатор и возвращающую объект с картой. Заметьте, что мы использовали для заголовка карты (параметр `title`) принятый на вход параметр `indicator`. Также мы установили соответствие между параметром `hover_name` и столбцом `Country Name`. Этот параметр отвечает за заголовок всплывающего окна при наведении мышью на страну. Высота карты была ограничена 650 пикселями. Об остальных параметрах мы уже достаточно говорили раньше. Разве что мы изменили заголовок цветовой шкалы, воспользовавшись написанной функцией `multiline_indicator`:

```
def display_generic_map_chart(indicator):
    df = poverty[poverty['is_country']]
    fig = px.choropleth(df, locations='Country Code',
                        color=indicator,
                        title=indicator,
                        hover_name='Country Name',
                        color_continuous_scale='cividis',
                        animation_frame='year', height=650)
    fig.layout.geo.showframe = False
    fig.layout.geo.showcountries = True
    fig.layout.geo.projection.type = 'natural earth'
    fig.layout.geo.lataxis.range = [-53, 76]
    fig.layout.geo.lonaxis.range = [-137, 168]
    fig.layout.geo.landcolor = 'white'
    fig.layout.geo.bgcolor = '#E5ECF6'
    fig.layout.paper_bgcolor = '#E5ECF6'
    fig.layout.geo.countrycolor = 'gray'
    fig.layout.geo.coastlinecolor = 'gray'
    fig.layout.coloraxis.colorbar.title =
multiline_indicator(indicator)
    return fig
```

Запустив код в JupyterLab, вы сможете выбрать из выпадающего списка нужный вам показатель и проанализировать данные по нему. На рис. 7.6 представлено несколько вариантов отображения карты при выборе разных комбинаций индикатора и года.

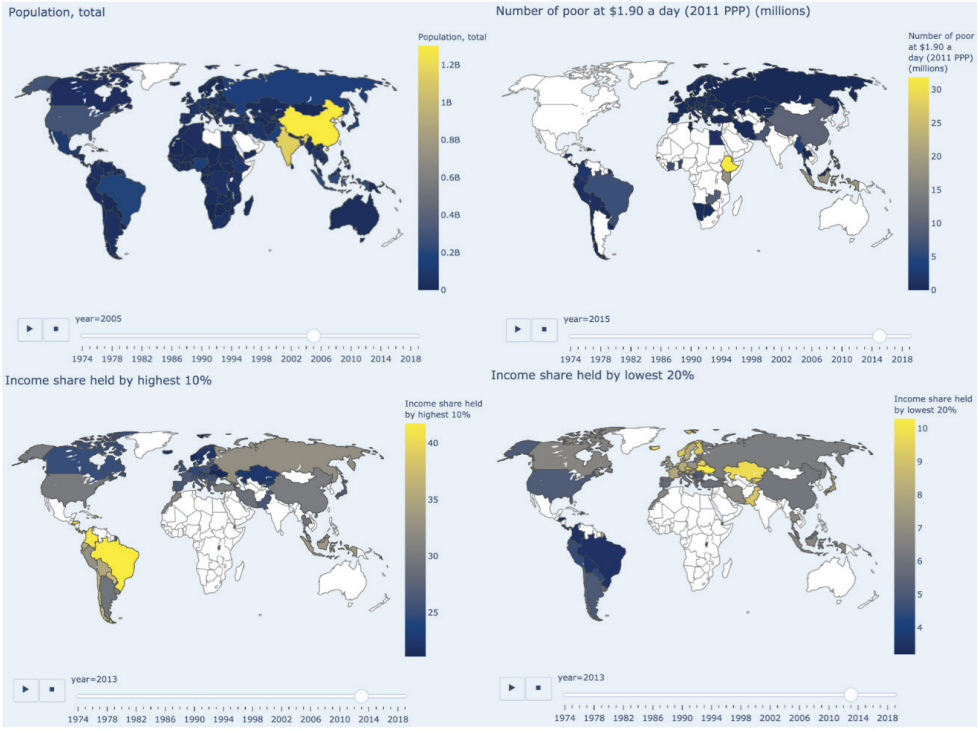


Рис. 7.6. Примеры отображения интерактивной карты

Обратите внимание, что пользователь может ввести несколько букв в поле выпадающего списка для поиска нужного ему индикатора. Но нам и пользователю может быть не понятно, что означают некоторые из индикаторов и какие у них есть ограничения. Было бы неплохо где-то выводить информацию об индикаторах, чтобы пользователю было легче определиться с выбором. Как мы уже упоминали, ограничения индикаторов имеют важное значение, и пользователь просто обязан знать о них. Давайте попробуем при помощи компонента `Markdown` вывести форматированный текст с необходимыми описаниями.

Создание компонента Markdown

Разметка или *Markdown* позволяют создавать код HTML простым и понятным способом, без использования традиционных тегов. Результат текста с разметкой будет выглядеть как обычный документ HTML, но на его создание уйдет гораздо меньше времени за счет облегченного синтаксиса. Сравните два приведенных ниже фрагмента кода, результатом которых является один и тот же код HTML.

С использованием традиционного подхода мы бы написали следующее:

```
<h1>This is the main text of the page</h1>
<h3>This is secondary text</h2>
```

```
<ul>
    <li>first item</li>
    <li>second item</li>
    <li>third item</li>
</ul>
```

А с применением языка разметки Markdown текст будет выглядеть так:

```
# This is the main text of the page
### This is secondary text
* first item
* second item
* third item
```

Думаю, вполне очевидно, что использование языка Markdown пошло на пользу читаемости исходника. Особенно заметно это при наличии множества вложенных тегов вроде <ul>.

Компонент Markdown работает по такому же принципу. Исходный текст должен быть передан ему в качестве параметра children, после чего будет сгенерирован код на языке HTML, показанный выше. Давайте создадим минимальное приложение в JupyterLab для демонстрации работы компонента Markdown.

1. Импортируем необходимые пакеты и создадим экземпляр приложения:

```
from jupyter_dash import JupyterDash
import dash_core_components as dcc
app = JupyterDash(__name__)
```

2. Создадим атрибут layout:

```
app.layout = html.Div([])
```

3. Поместим компонент Markdown в созданный элемент div и передадим ему на вход исходный текст. Заметьте, как удобно писать многострочный код с использованием тройных кавычек:

```
dcc.Markdown("""
# This is the main text of the page
### This is secondary text
* first item
* second item
* third item
""")
```

4. Запустите приложение:

```
app.run_server(mode='inline')
```

Вывод этого простого приложения показан на рис. 7.7.

This is the main text of the page

This is secondary text

- first item
- second item
- third item

Рис. 7.7. Вывод приложения с использованием компонента `Markdown`

Также с помощью языка разметки можно выводить нумерованные списки, таблицы, ссылки, жирный текст, курсив и многое другое. Мы еще несколько раз обратимся к языку разметки, но его синтаксис очень прост для понимания. Необходимо помнить, что существует несколько разновидностей языка Markdown на разных платформах, и вы могли встречать и другие варианты синтаксиса. Но в целом здесь есть очень много пересечений.

Итак, теперь давайте добавим в нашу секцию с интерактивной картой текстовые пояснения с использованием языка разметки. Всю основную информацию мы выведем под географической картой и слайдером, как показано на рис. 7.8.

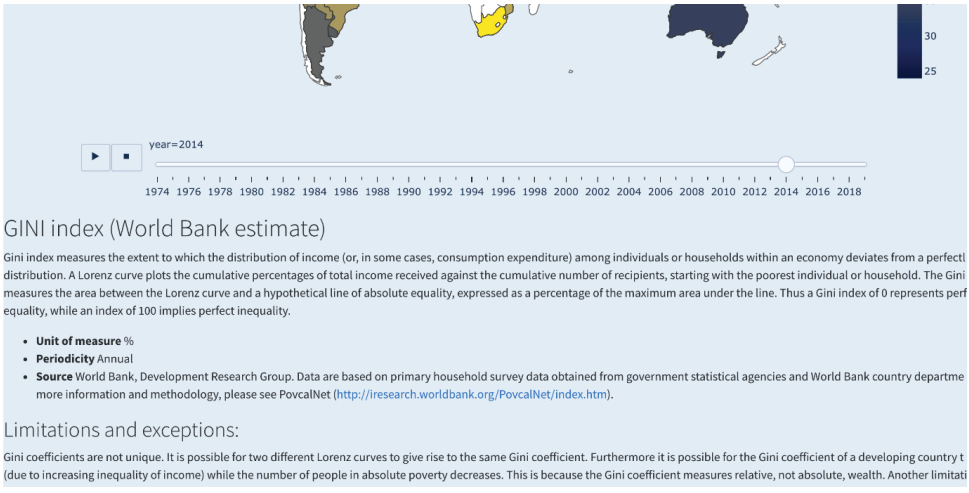


Рис. 7.8. Пример использования компонента `Markdown`

Весь текст и форматирование на рисунке выше реализованы с помощью языка разметки. Для создания разметки достаточно добавить компонент `Markdown` в области под картой и присвоить ему уникальный идентификатор.

Заполнение этого компонента будет производиться в той же функции обратного вызова, которая отвечает за построение карты. Мы добавим второй компонент `Output` нашей функции, и возвращать она будет уже не один, а два элемента. Для получения всей необходимой информации об индикаторах нам нужно открыть соответствующий файл. Мы уже делали это раньше, но повторим еще раз эту строчку кода: `series = pd.read_csv('data/PovStatsSeries.csv')`. Теперь давайте сделаем все оставшееся.

1. Под компонентом `Graph` добавим новый компонент `Markdown`. Заметьте, что для него мы также указали цвет фона приложения, чтобы все выглядело последовательно. Окончание `_md` в идентификаторе означает `Markdown`:

```
dcc.Markdown(id='indicator_map_details_md',
             style={'backgroundColor': '#E5ECF6'})
```

2. Обновим декоратор функции обратного вызова, добавив новый компонент:

```
@app.callback(Output('indicator_map_chart', 'figure'),
              Output('indicator_map_details_md', 'children'),
              Input('indicator_dropdown', 'value'))
```

3. После объявления переменной `fig` в функции обратного вызова создадим переменную `series`, выбрав строку с выбранным названием индикатора:

```
 series_df = series[series['Indicator Name'].eq(indicator)]
```

4. Извлечем содержимое столбца `Limitations and exceptions` из датафрейма `series_df`. Поскольку в данных есть пропуски и они не являются строками, мы заполним их значениями `N/A`, а последовательности из двух подряд символов переноса строки заменим на пробел. После этого получим первое значение из атрибута `values`:

```
limitations = series_df['Limitations and exceptions'].
fillna('N/A').str.replace('\n\n', ' ').values[0]
```

5. Теперь, когда мы создали переменные `series_df` и `limitations`, мы применим *f*-строки Python для форматирования нашей разметки. Сначала вставим имя индикатора в заголовок второго уровня `<h2>`. В языке разметки уровни заголовков обозначаются последовательностями символов решетки (`#`). Таким образом, создание тега `<h2>` в Markdown будет выглядеть так:

```
## {series_df['Indicator Name'].values[0]}
```

6. Добавим длинное описание индикатора без дополнительного форматирования:

```
{series_df['Long definition'].values[0]}
```

7. Создадим список для содержимого полей `Unit of measure`, `Periodicity` и `Source`. Элементы списка добавляются с помощью одиночной звездочки перед текстом. Обратите внимание, что мы попутно заменяем пропущенные значения в столбце `Unit of measure` на слово `'count'`, а в столбце `Periodicity` – на `'N/A'`. Две звездочки слева и справа от текста позволяют задать для него полужирный формат, как если бы мы написали `<b>text</b>`:

```
    * **Unit of measure** {series_df['Unit of measure'].
fillna('count').values[0]}
    * **Periodicity** {series_df['Periodicity'].fillna('N/A').
values[0]}
    * **Source** {series_df['Source'].values[0]}
```

8. Добавим заголовок третьего уровня:

```
### Limitations and exceptions:
```

9. Выведем содержимое переменной `limitations` в обычном формате:

```
{limitations}
```

Собрав весь приведенный выше код воедино, мы получим полное содержимое для нашего компонента `Markdown`. Заметьте, что у нас не всегда будет в наличии информация об индикаторах. В этом случае мы должны оповестить пользователя. Ниже показана такая проверка при помощи атрибута датафрейма `series_df.empty`:

```
...
fig.layout.coloraxis.colorbar.title =\
multiline_indicator(indicator)
series_df = series[series['Indicator Name'].eq(indicator)]

if series_df.empty:
    markdown = "No details available on this indicator"
else:
    limitations = series_df['Limitations and exceptions'].fillna('N/A').str.
replace('\n\n', ' ').values[0]
    markdown = f"""
    ## {series_df['Indicator Name'].values[0]}
    {series_df['Long definition'].values[0]}

    * **Unit of measure** {series_df['Unit of measure'].fillna('count').
values[0]}
    * **Periodicity** {series_df['Periodicity'].fillna('N/A').values[0]}
    * **Source** {series_df['Source'].values[0]}
    ### Limitations and exceptions:
    {limitations}
"""
return fig, markdown
```

Функция возвращает сразу две переменные `fig` и `markdown` вместо одной `fig`, как было раньше. Добавление этого кода в приложение позволит облегчить работу пользователю путем сообщения ему дополнительной информации и ограничений, которые необходимо учитывать при анализе.

Теперь мы познакомимся с тем, какие существуют проекции карт, и научимся выбирать нужную проекцию для отображения.

Знакомство с проекциями карты

До этого момента мы в наших примерах использовали только одну проекцию карты, а сейчас познакомимся с этой темой поближе. При попытке отобразить земной шар или какую-то его часть на плоскости мы не сможем избежать определенных искажений. Но мы можем использовать разные проекции, с которыми сейчас и познакомимся. Сразу скажем, что ни одна из проекций не идеальна, и речь может идти только о компромиссах в отношении точности фигур, их площадей, относительного расположения и т. д. Детали, связанные с выбором той или иной проекции карты, зависят от конкретного приложения, и их обсуждение выходит за рамки этой книги. Но мы узнаем, как можно посмотреть список доступных проекций и поменять одну на другую.

Функции построения карт Plotly Express располагают специальным параметром `projection`, принимающим на вход строку и позволяющим установить выбранную проекцию карты. Также проекцию можно установить, задав значение атрибуту `fig.layout.geo.projection.type` подобно тому, как мы делали это раньше.

На рис. 7.9 показаны некоторые из доступных проекций с их названиями в заголовках.

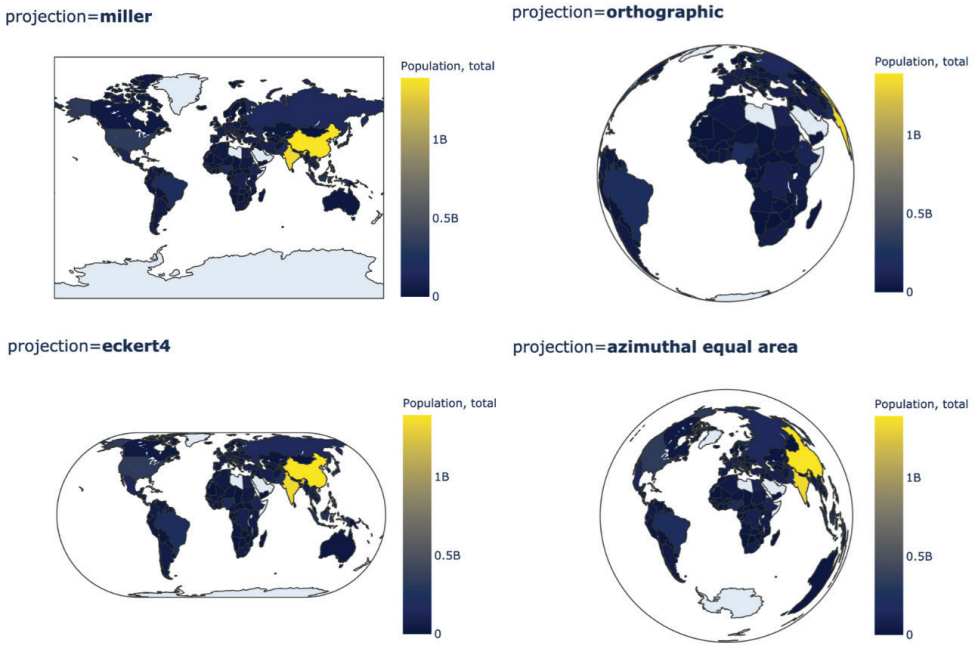


Рис. 7.9. Набор из доступных проекций карты

Как видите, нашу планету можно отобразить очень по-разному. И хотя *ортографическая* (orthographic) проекция выглядит наиболее привычно и реалис-

тично, проблема с ней заключается в невозможности одновременно увидеть все участки Земли. *Равновеликая азимутальная* (azimuthal equal area) проекция позволяет исключить этот недостаток, к тому же она выглядит довольно реалистично в интерактивном режиме при приближении карты. Вы можете поэкспериментировать с различными проекциями и выбрать наиболее подходящую вам.

Мы уже поработали с полигональными картами и картограммами, а теперь познакомимся поближе еще с одним типом карты, именуемым точечной картой.

Использование точечных карт

Главное отличие между осями *x* и *y*, с одной стороны, и широтой и долготой – с другой – состоит в кривизне земного шара. При приближении к экватору линии меридианов максимально удаляются друг от друга, тогда как на северном и южном полюсах сближаются. На рис. 7.10 это отчетливо видно.

Рис. 7.10. Карта земного шара с параллелями и меридианами

Иными словами, ближе к экватору площади, образованные параллелями и меридианами, приближаются к прямоугольной форме из-за того, что единица широты становится приблизительно равна единице долготы. Ближе к полюсам эти пропорции сильно меняются, и площади образуют фигуры, больше напоминающие треугольники. Это отличается от прямоугольной плоскости, где единица расстояния по вертикали равна единице расстояния по горизонтали вне зависимости от области плоскости, если применяется линейный масштаб по обеим осям. Исключением является логарифмическая ось, о которой мы говорили в главе 6. Проекции карты решают все эти вопросы, и с ними нам не нужно ни о чем беспокоиться. Вы можете просто думать о них как об осях *x* и *y* и выбирать проекцию исходя из своих требований.

Давайте посмотрим, как можно построить точечную карту в Plotly Express с использованием функции `scatter_geo`. Начнем с простого примера:

```
df = poverty[poverty['year'].eq(2010) & poverty['is_country']]
px.scatter_geo(df, locations='Country Code')
```

Сначала мы создали датафрейм, в котором ограничили данные 2010 годом и оставили только страны. Затем, как и в случае с картограммами, мы передали столбец параметру `locations`. Результат запуска этого кода показан на рис. 7.11.

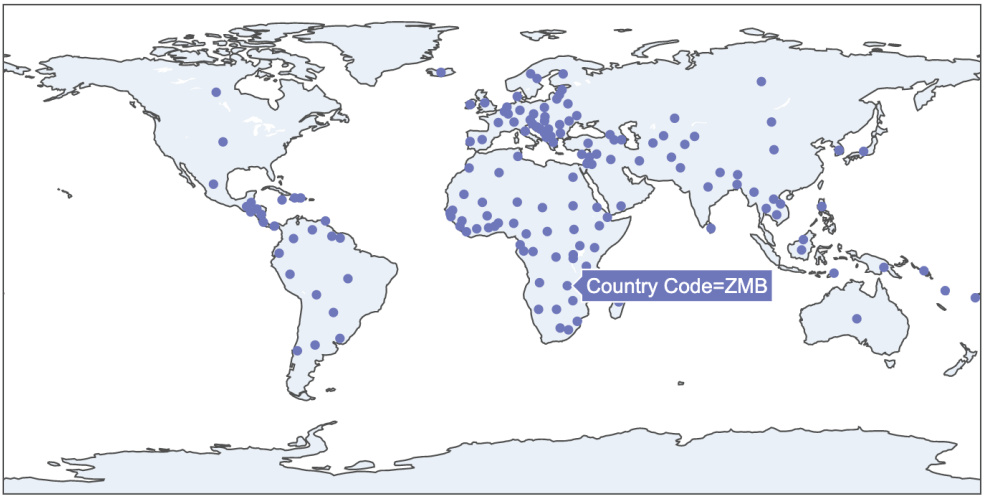


Рис. 7.11. Точечная карта, построенная с использованием функции `scatter_geo`

Как видите, все очень просто. На получившейся карте не так много информации, разве что маркеры на странах, отражающие их коды.

Названия стран поддерживаются в Plotly по умолчанию. Для задания произвольных точек на карте можно также использовать параметры `lat` и `lon`, как показано в следующем фрагменте кода и на рис. 7.12:

```
px.scatter_geo(lon=[10, 12, 15, 18], lat=[23, 28, 31, 40])
```

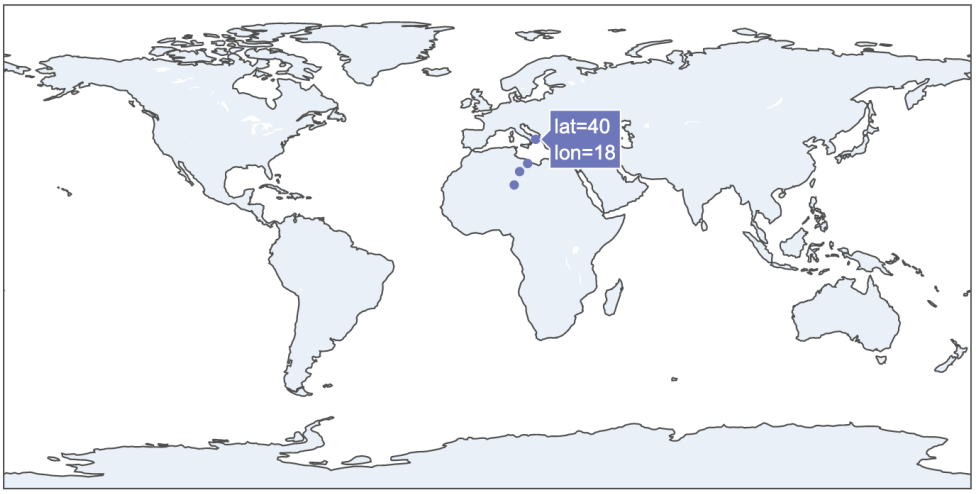


Рис. 7.12. Точечная карта с произвольными координатами

Вы можете воспользоваться всеми концепциями для изменения цвета, размера и прозрачности маркеров, которые мы обсуждали в главе 6.

Теперь рассмотрим еще один вариант создания географических карт – с использованием библиотеки Mapbox.

Использование карт Mapbox

`Mapbox` представляет собой библиотеку для построения карт с открытым исходным кодом. Ее поддержкой занимается одноименная компания, которая также поставляет расширения для библиотеки в виде дополнительных служб, слоев и тем. Примеры, которые мы будем рассматривать в этом разделе, могут быть легко запущены в Plotly, но есть и более продвинутые стили и службы, для использования которых нужно регистрировать аккаунт и применять токен каждый раз при построении карт.

Пример ниже поможет вам легко понять, как происходит создание географических карт в библиотеке Mapbox, поскольку вы уже знакомы с точечными картами:

```
px.scatter_mapbox(lon=[5, 10, 15, 20],
                  lat=[10, 7, 18, 5],
                  zoom=2,
                  center={'lon': 5, 'lat': 10},
                  size=[5]*4,
                  color_discrete_sequence=['darkred'],
                  mapbox_style='stamen-watercolor')
```

Здесь все должно быть предельно понятно. Параметры `lon` и `lat` являются аналогами параметров `x` и `y` на точечных картах. Параметры `size` и `color_discrete_sequence` мы уже освещали. Новым для вас здесь является параметр `zoom`, значение которого установлено в двойку. Он может принимать значения от 0 (весь мир) до 22 (приближение до дома) включительно. Также вы видите, что можно легко установить точку с центром карты с помощью параметра `center`, и мы сделали это, указав координаты первой точки (5, 10). Наконец, параметр `mapbox_style` позволяет отображать карту в различных стилях. Выбранный нами стиль `stamen-watercolor` показывает карту в художественном стиле, показанном на рис. 7.13.

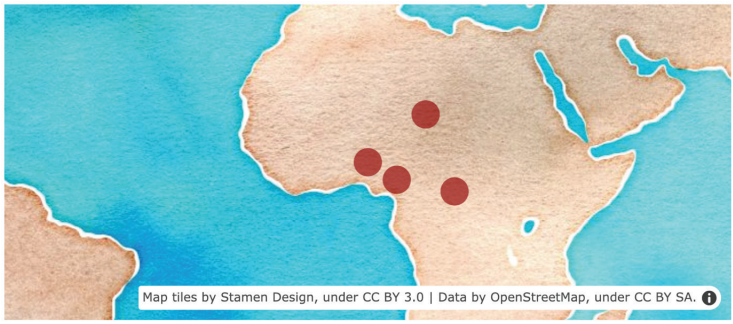


Рис. 7.13. Точечная карта с использованием библиотеки Mapbox и пользовательского стиля

Если навести на иконку с буквой **i** в правом нижнем углу карты, можно увидеть источники карты и данных. Как видите, в этой небольшой функции собрано довольно много слоев. Теперь давайте воспользуемся ей для вывода на карту данных из нашего набора.

Поскольку функция `scatter_mapbox` главным образом работает с данными о широте и долготе, а в нашем наборе они отсутствуют, мы сначала получим их, объединим с нашим датафреймом и только после этого выведем маркеры на карту.

Существует немало источников, в которых можно взять географические данные, и найти их в интернете не составит труда. Для извлечения нужных нам сведений мы воспользуемся функцией `read_html`. Она принимает на вход адрес URL, загружает список объектов `<table>` и возвращает их в виде списка датафреймов. После этого останется выбрать нужный. В следующем фрагменте кода мы извлекаем нужные нам сведения и сохраняем их в переменной `lat_long`:

```
lat_long = pd.read_html('https://developers.google.com/public-data/docs/
canonical/countries_csv')[0]
```

Далее мы воспользуемся знаниями, полученными в главе 4, и объединим датафреймы `lat_long` и `poverty` с помощью метода `merge`.

Сначала взглянем на структуру датафрейма `lat_long` в JupyterLab, выведя первые и последние пять строк, как показано на рис. 7.14.

	country	latitude	longitude	name
0	AD	42.546245	1.601554	Andorra
1	AE	23.424076	53.847818	United Arab Emirates
2	AF	33.939110	67.709953	Afghanistan
3	AG	17.060816	-61.796428	Antigua and Barbuda
4	AI	18.220554	-63.068615	Anguilla
...	...	...	...	...
240	YE	15.552727	48.516388	Yemen
241	YT	-12.827500	45.166244	Mayotte
242	ZA	-30.559482	22.937506	South Africa
243	ZM	-13.133897	27.849332	Zambia
244	ZW	-19.015438	29.154857	Zimbabwe

245 rows × 4 columns

Рис. 7.14. Датафрейм `lat_long` содержит широту и долготу по странам

В столбце **country** содержится двухбуквенный код страны. В нашем датафрейме `poverty` также присутствует колонка с этими кодами, которая называется **2-alpha code**. Соответственно, эти колонки можно использовать для объединения датафреймов, как показано ниже:

```
poverty = pd.merge(left=poverty, right=lat_long, how='left',
                   left_on='2-alpha code', right_on='country')
```

В результате столбцы из датафрейма `lat_long` будут добавлены к датафрейму `poverty` с дублированием данных при необходимости. Мы выполнили объединение слева (`how='left'`), а значит, таблица, указанная слева (`left=poverty`), будет использоваться в качестве основы. На рис. 7.15 показаны выборочные строки из объединенного датафрейма со странами и координатами.

	Country Name	longitude	latitude	2-alpha code
5849	Paraguay	-58.443832	-23.442503	PY
2755	Greece	21.824312	39.074208	GR
7145	Tajikistan	71.276093	38.861034	TJ
554	Belize	-88.497650	17.189877	BZ
6958	Suriname	-56.027783	3.919305	SR
6895	Sub-Saharan Africa	NaN	NaN	ZG
467	Belarus	27.953389	53.709807	BY
5828	Papua New Guinea	143.955550	-6.314993	PG
7159	Tajikistan	71.276093	38.861034	TJ
6850	St. Lucia	-60.978893	13.909444	LC

Рис. 7.15. Фрагмент объединенного датафрейма с широтой и долготой

Обратите внимание, что у стран, для которых не нашлось соответствия в датафрейме `lat_long`, в столбцах с координатами стоят значения **NaN**. Для стран, присутствующих в таблице более одного раза, например **Tajikistan**, значения координаты продублировались, чтобы можно было извлечь их из любой строки.

Итак, мы готовы к построению пузырьковой диаграммы (разновидности точечной, в которой с помощью размера маркеров отображается определенная переменная). Нам только нужно ограничить датафрейм странами и удалить строки с пропущенными значениями по выбранному индикатору (`Population, total`). Это можно сделать следующим образом:

```
df = poverty[poverty['is_country']].dropna(subset=['Population, total'])
```

Построить пузырьковую диаграмму можно при помощи функции `scatter_mapbox`, и мы пройдемся по каждому аргументу отдельно.

1. Вызовите функцию, передав ей созданный только что датафрейм:

   ```
   px.scatter_mapbox(df, ...)
   ```

2. Укажите колонки, которые будут использоваться для значений широты и долготы:

   ```
   lon='longitude', lat='latitude'
   ```

3. Установите желаемый уровень приближения, чтобы был виден весь земной шар:

```
zoom=1
```

4. Поставьте в соответствие значения индикатора размеру маркеров и установите подходящий максимальный размер:

```
size='Population, total', size_max=80
```

5. Свяжите показатель уровня дохода по странам с цветом маркеров (в нашем случае это дискретная переменная):

```
color='Income Group'
```

6. Выберите колонку `year` для выполнения анимации:

```
animation_frame='year'
```

7. Установите подходящий уровень прозрачности с учетом предполагаемого наложения маркеров:

```
opacity=0.7
```

8. Задайте высоту графика в пикселях:

```
height=650
```

9. Дополните информацию во всплывающем окне, добавив столбцы с уровнем дохода и регионом:

```
hover_data=['Income Group', 'Region']
```

10. Выберите подходящую цветовую последовательность для контроля различий в уровнях дохода по странам:

```
color_discrete_sequence=px.colors.qualitative.G10
```

11. Установите пользовательский стиль для карты:

```
mapbox_style='stamen-toner'
```

12. Установите заголовок для всплывающего окна с использованием названия страны:

```
hover_name=df['Country Name']
```

13. Задайте заголовок диаграммы:

```
title="Population by Country"
```

Выполнение этого кода, представленного ниже, приведет к выводу карты, показанной на рис. 7.16:

```
px.scatter_mapbox(df,
                  lon='longitude',
                  lat='latitude',
                  zoom=1,
                  size='Population, total',
                  size_max=80,
                  color='Income Group',
                  animation_frame='year',
                  opacity=0.7,
                  height=650,
                  hover_data=['Income Group', 'Region'],
                  color_discrete_sequence=px.colors.qualitative.G10,
                  mapbox_style='stamen-toner',
                  hover_name=df['Country Name'],
                  title='Population by Country')
```

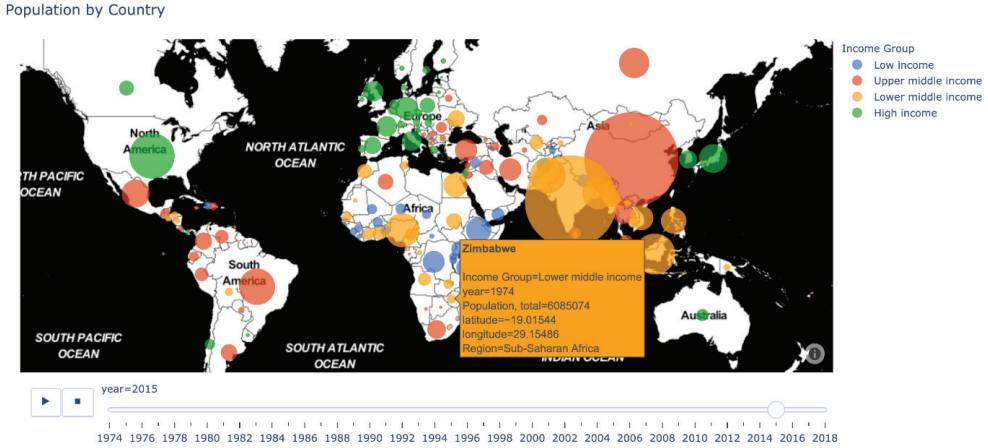


Рис. 7.16. Пузырьковая карта по численности населения по странам с динамикой по годам

Как видите, для построения довольно сложной карты нам потребовалось вызвать всего одну функцию с рядом простых и понятных параметров. Нужно только знать, что означает каждый из них и какие значения принимает.

Поскольку наша карта обладает интерактивностью, пользователь может менять уровень приближения, чтобы избежать наложения маркеров. На рис. 7.17 показана та же карта, но с приближением.

Одним из преимуществ пузырьковой карты над картограммой является то, что она позволяет показать, как определенные значения соотносятся с географической площадью стран (или других областей). К примеру, на рис. 7.16 хорошо заметно, что в Канаде, России и Австралии численность населения достаточно низкая относительно площади этих стран. Иными словами, перечисленные страны характеризуются низкой плотностью населения.

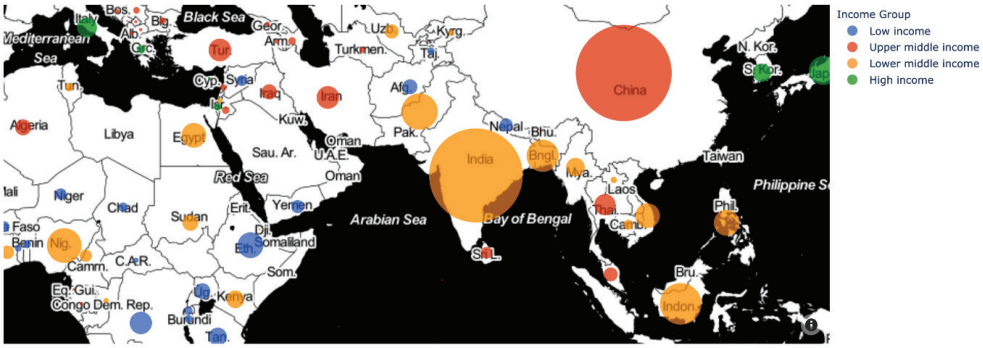


Рис. 7.17. Карта после увеличения выбранной области

Как видите, существует немало способов отображения географических карт и осуществления взаимодействий с ними, и мы не затронули даже малой части того, что вам доступно. Сейчас мы перечислим некоторые дополнительные возможности, применимые к географическим картам.

Другие опции и инструменты для работы с картами

В этом разделе мы кратко, не вдаваясь в подробности, рассмотрим опции, доступные вам при работе с картами.

Наверняка вы успели задуматься о том, чтобы выводить на географической карте произвольные полигоны, не ограничиваясь очертаниями стран. И у вас есть подобная возможность.

Для представления такой информации существует стандартный формат `GeoJSON`. В основном он состоит из точек, линий и полигонов. Точки представляют собой просто местоположение на карте, подобно тому, что мы видели при работе с точечными картами. Линии – это группы соединенных точек в определенной последовательности, при этом первая точка не совпадает с последней. И, как вы уже догадались, полигон – это линия с совпадающими координатами первой и последней точек. Обратите внимание, что многие страны описываются далеко не единственным полигоном. Большинство функций Plotly для работы с картами поддерживают формат GeoJSON, и вы можете использовать его для своих нужд.

Это бывает полезно в случаях, когда у вас есть пользовательские данные о пользовательских местоположениях и вам необходимо извлечь для них сопутствующую информацию.

Также стоит упомянуть об очень любопытном проекте `geopandas`. Как ясно из названия, эта специализированная библиотека работает по принципам `pandas`, но с предоставлением особых структур данных и техник, направленных на взаимодействие с географическими сведениями. Одной из таких структур является `GeoDataFrame`. Вам стоит выделить время для ознакомления с этими инструментами, если вы собираетесь плотно работать с картами и преобразовывать их.

Что ж, пришло время перенести созданный функционал в наше приложение.

Внедрение интерактивной карты в приложение

Созданная нами карта с выпадающим списком и сопутствующей разметкой имеет все основания стать первым исследовательским инструментом в нашем приложении. Мы можем удалить столбчатую диаграмму с численностью населения, а на ее место поместить созданные только что компоненты, чтобы пользователь мог выбирать и исследовать нужные ему индикаторы, выводя их на карту и перемещаясь по годам. Попутно для каждого выбранного индикатора ему будет доступно полное описание и сопутствующие ограничения. Заинтересовавшись конкретным показателем, пользователь сможет построить по нему другую диаграмму с более подробной информацией.

Для переноса созданного функционала в наше приложение нужно выполнить следующие действия.

1. Добавьте объявление датафрейма `series` в начало файла `app.py`:

```
series = pd.read_csv('data/PovStatsSeries.csv')
```

2. Добавьте определение функции `multiline_indicator` до размещения определения макета `app.layout`:

```
def multiline_indicator(indicator):
    final = []
    split = indicator.split()
    for i in range(0, len(split), 3):
        final.append(' '.join(split[i:i+3]))
    return '<br>'.join(final)
```

3. Добавьте компоненты `Dropdown`, `Graph` и `Markdown` в начало файла, под основными заголовками, где раньше размещалась столбчатая диаграмма. В приведенном ниже коде показано, как это сделать, включая указание идентификаторов компонентов, но остальные параметры мы опустили. Обратите внимание на появление компонента `Col` и установку ширины другого компонента `Col` с использованием параметра `lg`. Первый используется для вывода пустой колонки перед отображением содержимого карты, а второй отвечает за само графическое содержимое визуализации:

```
app.layout = html.Div([
    dbc.Col([
        html.Br(),
        html.H1('Poverty And Equity Database'),
        html.H2('The World Bank'),
    ], style={'textAlign': 'center'}),
    html.Br(),
    dbc.Row([
```

```
        dbc.Col(lg=2),
        dbc.Col([
            dcc.Dropdown(id='indicator_dropdown', ...),
            dcc.Graph(id='indicator_map_chart', ...),
            dcc.Markdown(id='indicator_map_details_md', ...)
    ], lg=8)
    ]),
    html.Br()
```

В этой главе мы познакомились с некоторыми полезными инструментами, так что давайте подведем некоторые итоги.

Заключение

Эту главу мы начали со знакомства с картограммами, представляющими особый тип географических карт, к которым мы все привыкли. Также мы посмотрели, как можно анимировать карты с использованием дополнительного последовательного параметра. В нашем случае мы применили анимацию выбранного индикатора по годам. После этого мы создали функцию обратного вызова, дав возможность пользователю работать с использованием карт со всеми нужными ему индикаторами.

Далее мы познакомились с языком разметки Markdown для генерирования содержимого в формате HTML и узнали, как внедрить его в приложение Dash. Попутно мы освоили разные варианты отображения карт и поработали с различными проекциями.

После этого мы приступили к изучению еще одного типа карт – точечной карты. Вооружившись знаниями, полученными в предыдущих главах, мы смогли с легкостью применить их к этому виду визуализации. Также мы познакомились с богатым функционалом специализированной библиотеки Mapbox и другими ключевыми возможностями при отображении карт. Наконец, мы интегрировали новую секцию в наше приложение, в котором на данный момент присутствует вся необходимая информация для исследования нужных пользователю индикаторов.

В следующей главе мы поработаем с принципиально другим типом графиков, показывающим характер распределения значений в последовательности. Этот график называется **гистограммой**. Также мы познакомимся с еще одним важным компонентом Dash – **DataTable**, позволяющим выводить табличные данные в улучшенном виде, визуализировать их, загружать и т. д.

Глава 8

Определение частотности данных с помощью гистограмм и построение интерактивных таблиц

Все типы диаграмм, которые мы изучали до сих пор, отображали сами данные как они есть. Иными словами, каждый маркер, будь то круг, столбик, символ или что-то еще, относился к конкретной точке данных в наборе. *Гистограммы* (histogram) же выводят столбики, соответствующие суммарной статистике о группах данных. Гистограмма главным образом используется для подсчета значений в наборе данных. Это делается при помощи объединения значений в группы или столбики и вывода количества наблюдений в каждой группе. Конечно, можно выводить и другую статистическую информацию о группах, например средние или максимальные значения, но в большинстве случаев используется именно подсчет количества.

При этом количество элементов в группе визуализируется при помощи столбика, а сам этот показатель определяется высотой столбика. Здесь важно отметить, что подобный вид визуализации позволяет быстро и довольно точно составить мнение о распределении значений в наборе данных. Собрано ли большинство значений в одной или нескольких точках, наблюдается ли перекос в левую или правую сторону? Ответы на эти вопросы позволяют достаточно четко понять профиль исходных данных.

Распределение вероятностей – это основа статистики, позволяющая понять глубинные характеристики данных. Для понимания сущности данных критически важно знать, как именно распределены значения. Если они распределены нормально, мы сможем делать одни выводы и строить соответствующие ожидания, а если экспоненциально – совсем другие. Гистограмма идеально подходит для визуализации частотных характеристик данных.

Помимо гистограмм, мы также познакомимся в этой главе с компонентом **DataTable**. Это очень гибкий и мощный инструмент, позволяющий отображать, фильтровать и экспортировать таблицы с данными.

Давайте перечислим темы, которые будут рассмотрены в главе:

- создание гистограммы;
- настройка гистограммы, включая изменение количества столбиков и отображение множественных данных;
- добавление гистограммам интерактивности;
- создание двумерной гистограммы;
- создание DataTable;
- настройка отображения таблицы данных (ширина и высота ячеек, отображение текста и т. д.);
- добавление гистограмм и таблиц данных в приложение.

Технические требования

В данной главе мы продолжим использовать инструменты и пакеты, с которыми познакомились в предыдущих главах, с одним дополнением. Для создания графиков мы продолжим использовать Plotly Express и модуль `graph_objects`. Пакеты, которые мы будем применять: Plotly, Dash, Dash Core Components, Dash HTML Components, Dash Bootstrap Components, pandas, а также новый пакет dash_table. Вам нет необходимости устанавливать его отдельно (хотя вы можете это сделать), поскольку он устанавливается вместе с Dash.

Исходный код к этой главе располагается в хранилище GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_08**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3sGSCes**.

Создание гистограммы

Посмотрим, как можно максимально просто получить полную информацию о распределении значений в наборе данных и узнать характер и степень их изменчивости. На помощь нам придет гистограмма.

Как и всегда, начнем с простейшего примера.

1. Откроем датафрейм `poverty` и ограничим его только странами и 2015 годом, как показано ниже:

```
import pandas as pd
poverty = pd.read_csv('data/poverty.csv')
df = poverty[poverty['is_country'] & poverty['year'].
eq(2015)]
```

2. Импортируем модуль Plotly Express и вызовем функцию `histogram`, передав в качестве параметра `data_frame` наш датафрейм, а в качестве параметра `x` – индикатор для анализа:

```
import plotly.express as px
gini = 'GINI index (World Bank estimate)'
px.histogram(data_frame=df, x=gini)
```

В результате будет выведена гистограмма, показанная на рис. 8.1.

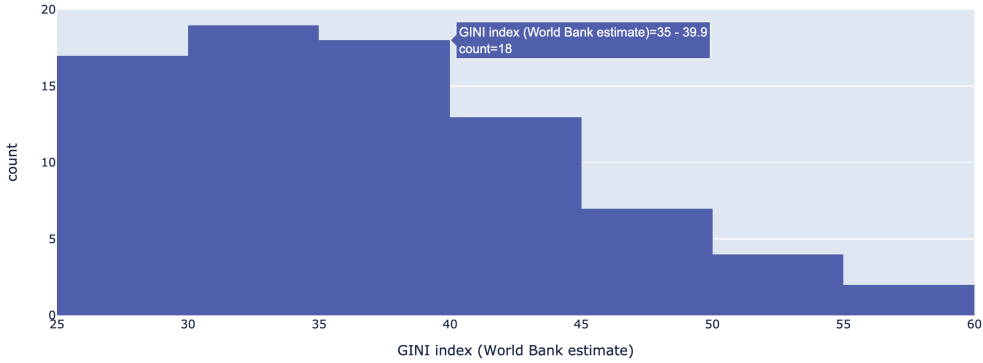


Рис. 8.1. Гистограмма с индексом Джини

Ось `x` получила в качестве подписи название индикатора, а `y` была названа `count` по имени метода, который функция использует для агрегации по умолчанию. Это видно и во всплывающем окне, появляющемся при наведении мышью на столбики. На рис. 8.1 показано всплывающее окно для столбика в интервале (35, 39.9), и стран с индексом, попадающим в этот диапазон в 2015 году, было 18. Ранее мы уже визуализировали этот индикатор отдельно по странам, а теперь пытаемся взглянуть на него под другим углом и определить распределение его значений. Мы видим, что большинство стран характеризуются индексом Джини в интервале от 25 до 40, а с увеличением значения этого показателя количество стран заметно уменьшается. Конечно, это характерно для конкретного выбранного года.

В данный момент мы использовали количество столбиков на гистограмме по умолчанию, но вы легко можете изменить это значение, пока не получите удовлетворительный результат. Также вы можете позволить пользователю самому устанавливать желаемое количество столбиков, особенно если заранее неизвестно, какой именно индикатор он будет анализировать при помощи гистограммы. И это как раз наш случай.

Давайте взглянем, каким будет эффект от изменения количества столбиков на гистограмме и модификации других доступных параметров.

Настройка гистограммы, включая изменение количества столбиков и отображение множественных данных

Изменить количество отображаемых столбиков на гистограмме можно при помощи параметра `nbins`. Сначала посмотрим, как будет выглядеть график с двумя экстремальными значениями. Если установить параметру `nbins` значение 2, гистограмма будет выглядеть так, как на рис. 8.2.

Как видите, все значения индикатора были разбиты на две группы в интервалах (20, 39.9) и (40, 59.9), и мы видим, сколько стран присутствует в каждой из этих групп. Здесь есть о чем подумать, но информация, согласитесь, получилась не столь красноречивой, как на рис. 8.1. Теперь попробуем задать значение `nbins=500`. Результат показан на рис. 8.3.

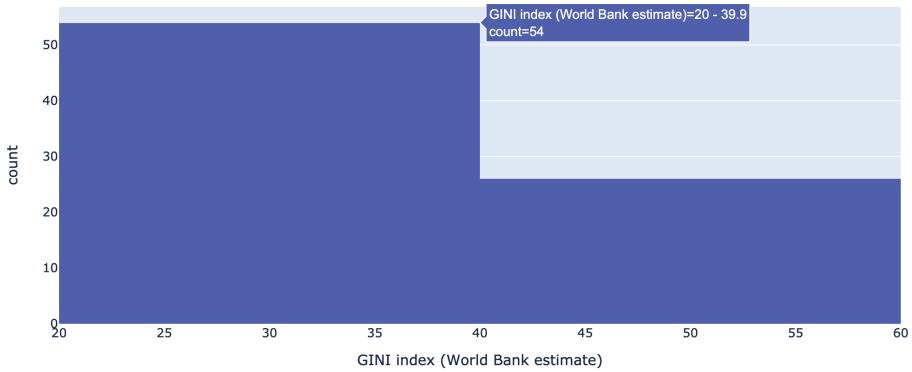


Рис. 8.2. Гистограмма с двумя столбиками

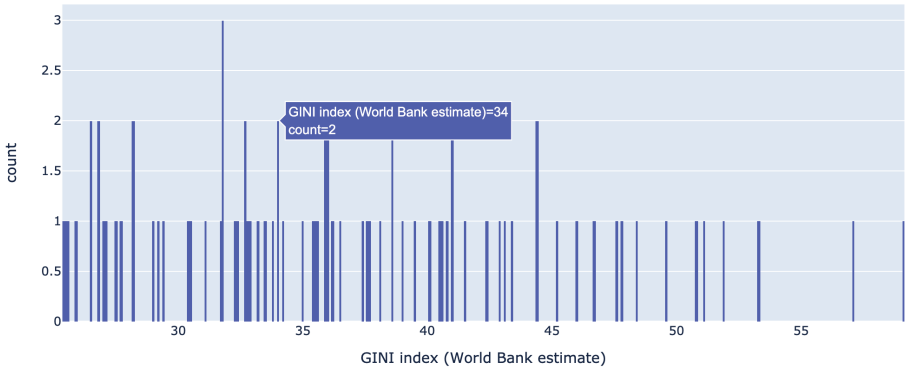


Рис. 8.3. Гистограмма с 500 столбиками

Теперь диаграмма оказалась более детализированной – даже больше, чем нужно. При разбиении данных на большое количество столбиков они приобретают вид, приближенный к исходным показателям.

По умолчанию на гистограмме было пять столбиков. Теперь, когда мы знаем, что наши значения лежат в интервале от 25 до 60 (45 единиц), мы можем посмотреть, как они распределятся по 45 столбикам. На рис. 8.4 показан вывод.

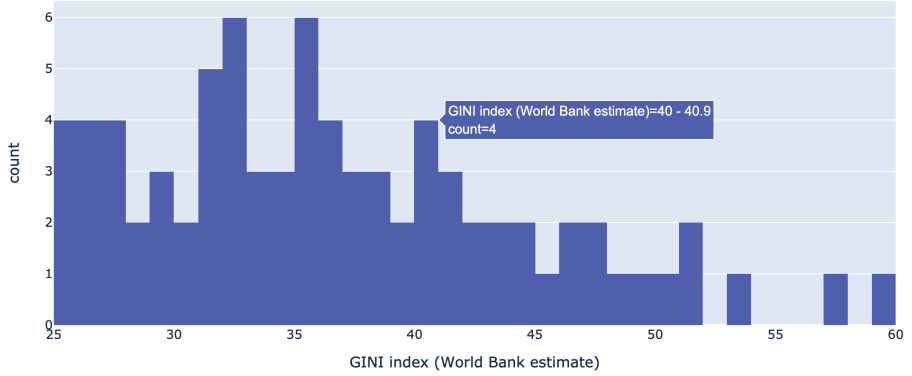


Рис. 8.4. Гистограмма с 45 столбиками

Все графики, которые мы построили в этом разделе, описывали одни и те же данные. Вы видите, как сильно меняется картина при выборе разных значений для количества столбиков. Это можно воспринимать как взгляд на распределение значений с разным разрешением. Обычно существует какое-то идеальное разрешение для каждого отдельного случая, которое можно найти методом проб и ошибок. Именно это делает интерактивные гистограммы, в которых пользователь может сам менять количество столбиков для отображения, столь полезными.

Вспомните, что у нас в наборе данных присутствуют несколько категориальных столбцов, которые мы могли бы использовать для установки цвета наших колонок с целью получения дополнительной информации о данных. Давайте посмотрим, как это можно сделать.

Использование цвета для детализации гистограммы

Как вы уже догадались, добавить цвет на гистограмму в Plotly Express можно очень просто – для этого достаточно указать колонку из датафрейма, которая будет отвечать за цветовую составляющую. К примеру, установка параметра `color='Income Group'` приведет к отрисовке графика, показанного на рис. 8.5.

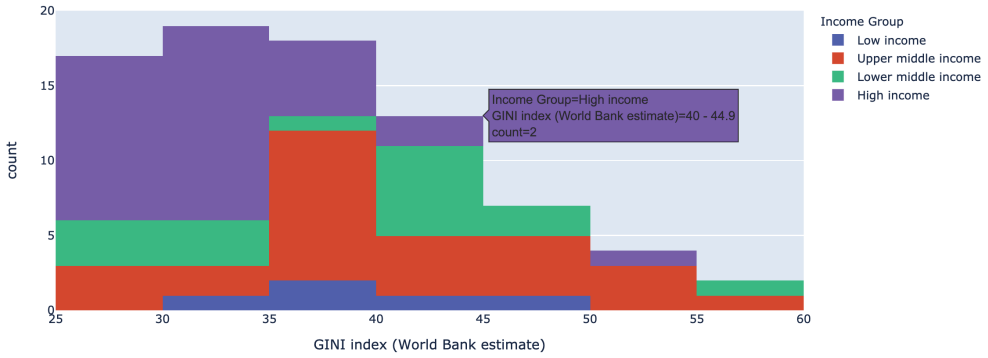


Рис. 8.5. Гистограмма по индексу Джини с цветовым разделением по уровням дохода

Это та же самая гистограмма, но обогащенная при помощи еще одного разреза аналитики. В результате каждый столбик поделен по цветам, соответствующим значениям из колонки **Income Group**. В результате мы можем видеть, как страны распределяются по уровню дохода в рамках одного сегмента интервала индекса Джини.

На рис. 8.6 показан результат применения следующих параметров: `color='Region', color_discrete_sequence=px.colors.qualitative.Set1`.

Опять же, контуры самой гистограммы не изменились, а изменилось только цветовое наполнение. Теперь оно указывает на регионы. Если помните, в главе 5 при обсуждении столбчатых диаграмм мы говорили, что существует несколько подходов к отображению столбиков. Также настройку внешнего вида гистограммы можно выполнять при помощи параметра `barmode`. Давайте посмотрим, какие варианты нам доступны.

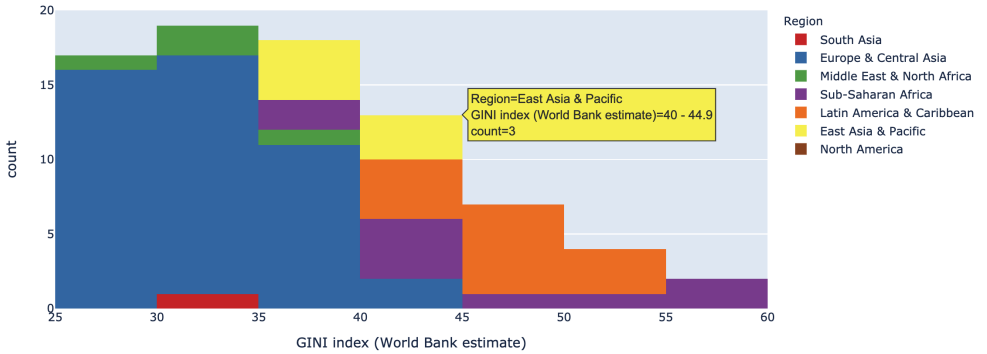


Рис. 8.6. Гистограмма по индексу Джини с цветовым разделением по регионам

> **Совет**
>
> Вы могли заметить, что на гистограммах столбцы располагаются вплотную друг к другу, без зазоров, как в случае со столбчатыми диаграммами. Это сделано специально – для лучшего понимания самой природы гистограммы. Столбики здесь фактически отделяют одни наблюдения от других в общей массе. Как вы видели, интервалы могут быть установлены разные, и в зависимости от этого будет меняться внешний вид диаграммы. Столбчатые диаграммы, напротив, чаще используются для анализа различий между элементами категориальной переменной, и разрывы между столбиками прямо указывают на эту особенность.

Отображение множественных гистограмм

На показанных выше гистограммах мы видели, что каждый столбик был разбит на несколько секций по выбранному параметру группировки. Эти секции фактически демонстрировали распределение показателя внутри интервала в зависимости от группы.

Но бывают случаи, когда нам, к примеру, нужно показать распределение показателя за два разных года. Здесь расположение данных друг над другом в одном столбике может быть неправильно воспринято. Может создаться ложное ощущение того, что эти сегменты являются частью целого, хотя на самом деле они относятся к разным годам. Продемонстрируем это на примере.

1. Создадим датафрейм на основе `poverty` с данными по 2010 и 2015 годам только по странам:

```
df = poverty[poverty['is_country'] & poverty['year'].
isin([2010, 2015])]
```

2. Вызовем функцию `histogram` для индекса Джини с распределением цветов по годам и установкой параметра `barmode='group'`:

```
px.histogram(df, x=gini, color='year', barmode='group')
```

В результате получится график, показанный на рис. 8.7.

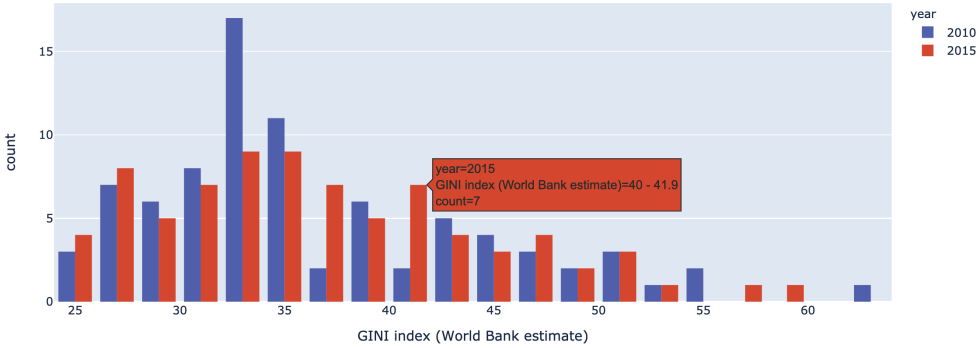


Рис. 8.7. Гистограмма по индексу Джини с установкой параметра `barmode='group'`

Поскольку информация по годам представляется нам как «до и после», имеет смысл располагать столбики рядом хронологически, чтобы можно было понять, как меняется распределение показателя в зависимости от года.

Существует еще один подход к отображению множественных гистограмм, он применим тогда, когда нам необходимо сравнить само распределение за разные годы. Для этого можно вызвать ту же функцию, но в добавление к распределению по цветам использовать фасеты для разделения гистограммы на два разных графика. Это делается следующим образом:

```
px.histogram(df, x=gini, color='year', facet_col='year')
```

Результат показан на рис. 8.8.

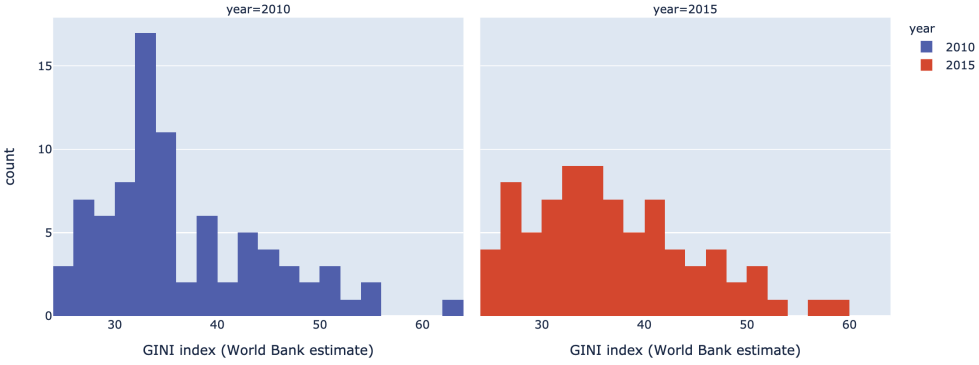


Рис. 8.8. Гистограмма по индексу Джини с разбиением по годам

Опять же, последние графики показывают одну и ту же информацию двумя разными способами. На гистограмме, представленной на рис. 8.7, можно легко отслеживать динамику изменения количества стран в группах от года к году. В то же время труднее понять, как изменился сам вид распределения. На

рис. 8.8 именно этот аспект отражен прекрасно. Заметьте, что можно вместо параметра `facet_col` использовать `facet_row` – это позволит расположить графики не бок о бок, а друг над другом. Мы выбрали горизонтальную ориентацию для отображения графиков, поскольку нам интересно сравнить высоты столбиков для разных лет.

Также иногда нас больше интересует процентное распределение элементов в общей группе, а не их абсолютное количество. Построить такой график очень просто. Для этого достаточно передать функции параметр `histnorm='percent'`. Начнем с создания переменной `fig` и добавления новой опции:

```
fig = px.histogram(df, x=gini, color='year', facet_col='year',
histnorm='percent')
```

Мы можем более явно показать, что используем именно процентное распределение, добавив знак процента к подписям меток на оси *y*. Это можно сделать так:

```
fig.layout.yaxis.ticksuffix = '%'
```

Также можно снабдить вертикальную ось внятной подписью следующим образом:

```
fig.layout.yaxis.title = 'Percent of total'
```

Запуск измененного кода приведет к отображению графика, показанного на рис. 8.9.

Рис. 8.9. Процентная гистограмма по индексу Джини с разбиением по годам

Этот график выглядит так же, как и предыдущий. Главным отличием между ними является то, что высота столбиков в данном случае отражает процент стран в каждой группе, а не их количество. Это также видно по подписям и заголовку вертикальной оси.

Мы рассмотрели несколько опций, применимых к гистограммам, а теперь давайте добавим им интерактивности.

Добавление гистограммам интерактивности

Гистограммы могут быть интерактивными, как и любые другие виды диаграмм. Пользователи вольны сами определять, какие показатели и за какие годы анализировать. К тому же в случае с гистограммами мы можем дать пользователю возможность выбрать количество столбиков для отображения. Мы уже умеем писать функции обратного вызова, обрабатывающие несколько элементов ввода и вывода, и здесь мы этим воспользуемся. На рис. 8.10 показана визуализация, к которой мы будем стремиться.

Рис. 8.10. Приложение с интерактивными гистограммами

При написании кода мы не будем подробно обсуждать некоторые компоненты на макете, вы всегда можете посмотреть готовое решение в репозитории. Мы же главным образом сконцентрируемся на нюансах, связанных с интерактивностью компонентов. После этого добавим новый функционал в наше приложение.

1. Объявим блок импортирования необходимых пакетов:

```
from jupyter_dash import JupyterDash
import dash_core_components as dcc
import dash_html_components as html
import dash_bootstrap_components as dbc
from dash.dependencies import Output, Input
```

2. Создадим объект app и макет layout:

```
app = JupyterDash(__name__)
app.layout = html.Div([])
```

3. Добавим компоненты Label и Dropdown в наш элемент div. В выпадающем списке у нас будут находиться доступные для анализа индикаторы, мы уже делали это в главе 7:

```
html.Div([
    dbc.Label('Indicator:'),
    dcc.Dropdown(id='hist_indicator_dropdown', optionHeight=40,
                 value='GINI index (World Bank estimate)',
                 options=[{'label': indicator, 'value': indicator}
                          for indicator in poverty.columns[3:54]]),
])
```

4. Добавим компоненты Label и Dropdown для выбора пользователем одного или нескольких лет для анализа. Заметьте, что поскольку в выпадающем списке допускается множественный выбор, значение по умолчанию ему должно передаваться в виде списка:

```
dbc.Label('Years:'),
dcc.Dropdown(id='hist_multi_year_selector',
             multi=True,
             value=[2015],
             placeholder='Select one or more years',
             options=[{'label': year, 'value': year}
                 for year in poverty['year'].drop_duplicates().
sort_values()]),
```

5. В тот же элемент div добавим компоненты Label и Slider для управления количеством отображаемых столбиков. Если не задать значение по умолчанию для количества столбиков, Plotly вычислит его самостоятельно на основе данных. На слайдере в это время будет показана отметка 0, и пользователь сможет настроить ползунок по своему желанию:

```
dbc.Label('Modify number of bins:'),
dcc.Slider(id='hist_bins_slider',
           min=0, step=5,
           marks={x: str(x) for x in range(0, 105, 5)}),
```

6. Наконец, добавим компонент Graph и завершим наш макет:

```
dcc.Graph(id='indicator_year_histogram')
```

Если запустить приложение на этом этапе, мы получим только итоговый макет без интерактивности. Он будет выглядеть так, как показано на рис. 8.11.

Цвета, расположение элементов и их относительное позиционирование я оставляю вам на откуп, с этой задачей вы справитесь и сами, вооружившись знаниями, полученными в первой главе.

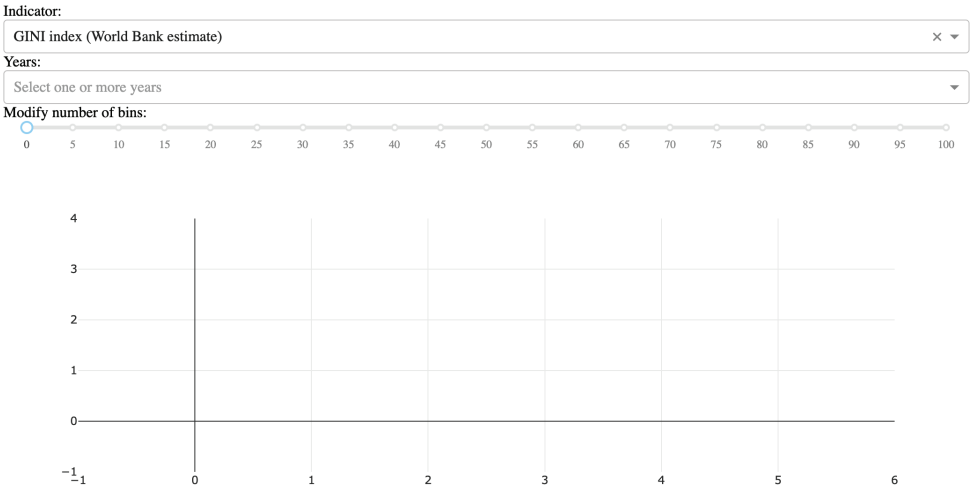


Рис. 8.11. Внешний вид приложения без интерактивности

Теперь мы сделаем следующий шаг и настроим взаимодействие между компонентами. Нам потребуется функция обратного вызова с тремя элементами ввода (выпадающие списки с индикаторами и годами, а также слайдер с количеством столбиков). Возвращать функция будет объект `Figure`, который пользователь будет видеть под элементами управления.

1. Создадим декоратор обратного вызова. Здесь ничего особенного, вы все это уже делали не раз:

```
@app.callback(Output('indicator_year_histogram', 'figure'),
              Input('hist_multi_year_selector', 'value'),
              Input('hist_indicator_dropdown', 'value'),
              Input('hist_bins_slider', 'value'))
```

2. Начнем писать функцию, которая будет строить гистограмму на основе переданных элементов ввода. Также сделаем проверку на то, что в обоих выпадающих списках выбраны значения:

```
def display_histogram(years, indicator, nbins):
    if (not years) or (not indicator):
        raise PreventUpdate
```

3. Создадим набор данных, выбрав из исходного датафрейма только страны, а также ограничив список только теми годами, которые были выбраны в выпадающем списке:

```
df = poverty[poverty['year'].isin(years) & poverty['is_country']]
```

4. Теперь у нас все готово для создания графика при помощи функции `histogram`. При этом мы воспользуемся следующими опциями: в качестве параметра `data_frame` передадим наш датафрейм `df`, параметр `x` будет принимать значение `indicator`, а `color` – `year`. Заголовок гистограммы мы соберем из названия выбранного индикатора и слова `Histogram`. Также в качестве дополнительного параметра `nbins` мы передадим элемент ввода `nbins`, в котором содержится выбранное пользователем количество столбиков на гистограмме. Для настройки фасетов используем столбец `year`. Поскольку мы заранее не знаем, сколько именно лет для анализа выберет пользователь, установим параметр `facet_col_wrap=4`. Это позволит ограничить вывод в одной строке четырьмя графиками, а следующие будут добавлены уже на новой строке:

```
fig = px.histogram(df, x=indicator, facet_col='year', color='year',
                        title=indicator + ' Histogram',
                        nbins=nbins,
                        facet_col_wrap=4, height=700)
```

5. Есть еще один любопытный атрибут у объекта `Figure`, которого мы раньше не касались, – это `for_each_xaxis`. Заметьте, что это лишь один из многих атрибутов, начинающихся с `for_each_`, которые вы можете использовать по отдельности. Данный атрибут удобно использовать в случаях, когда у вас есть несколько атрибутов `xaxis`. По умолчанию каждый график на отдельном фасете будет иметь собственный заголовок оси *x*. Но, как вы знаете, в нашем наборе данных присутствуют очень длинные названия индикаторов, которые в этом случае будут накладываться один на другой. Чтобы избежать этого, мы присвоим всем заголовкам осей пустую строку, как показано ниже:

```
fig.for_each_xaxis(lambda axis: axis.update(title=''))
```

6. Взамен удаленным подписям к горизонтальным осям мы можем добавить на диаграмму аннотацию. *Аннотация* (annotation) – это просто строка, которую можно добавить при помощи метода `add_annotation`. Поскольку мы хотим, чтобы аннотация была отображена посередине графика, установим в качестве параметра `x` значение `0.5`. По вертикальной оси подпись должна располагаться чуть ниже гистограммы или набора гистограмм, так что для параметра `y` зададим значение `-0.12`. Также важно указать Plotly, откуда вести отсчет переданных координат аннотации. Для этого существуют параметры `xref` и `yref`, и мы установим им значение `paper` в качестве точки отсчета. Это будет означать, что координаты в относительном выражении будут отсчитываться от базовой точки рисунка в целом, а не от каждой точки данных, как, например, в случае с точечной диаграммой. Нам как раз и требуется, чтобы аннотация в виде названия индикатора имела фиксированную позицию под графиками. По умолчанию аннотация отображается со стрелкой, но нам стрелка не нужна, и избавиться от нее можно, установив параметр

showarrow=False. В целом инструкция по добавлению аннотации на график будет выглядеть так:

```
fig.add_annotation(text=indicator,
                   x=0.5, y=-0.12,
                   xref='paper', yref='paper', showarrow=False)
```

Ниже приведен полный код функции:

```
@app.callback(Output('indicator_year_histogram', 'figure'),
              Input('hist_multi_year_selector', 'value'),
              Input('hist_indicator_dropdown', 'value'),
              Input('hist_bins_slider', 'value'))
def display_histogram(years, indicator, nbins):
    if (not years) or (not indicator):
        raise PreventUpdate
    df = poverty[poverty['year'].isin(years) & poverty['is_country']]
    fig = px.histogram(df, x=indicator, facet_col='year', color='year',
                       title=indicator + ' Histogram',
                       nbins=nbins,
                       facet_col_wrap=4, height=700)
    fig.for_each_xaxis(lambda axis: axis.update(title=''))
    fig.add_annotation(text=indicator, x=0.5, y=-0.12,
                       xref='paper', yref='paper', showarrow=False)
    fig.layout.paper_bgcolor = '#E5ECF6'
    return fig
```

Мы написали приложение, которое можно запустить в JupyterLab. Проверьте его на предмет наличия ошибок и попытайтесь сами внести какие-нибудь доработки.

До сих пор мы визуализировали количество элементов и распределение значений только для одного набора наблюдений. Но существует также интересный способ исследовать одновременно два набора значений при помощи так называемой двумерной гистограммы.

Создание двумерной гистограммы

В предыдущем разделе мы занимались подсчетом наблюдений в каждом столбике на нашей гистограмме. Сейчас будем делать то же самое, но для комбинаций столбиков по двум наборам данных. В результате на пересечении столбики по двум наборам будут образовывать своеобразную матрицу. Простой пример все расставит по местам. Давайте построим *двумерную гистограмму* (2D histogram).

1. По традиции создадим поднабор данных на основе датафрейма poverty, содержащий только страны и ограниченный данными за 2000 год:

```
df = poverty[poverty['year'].eq(2000) & poverty['is_country']]
```

2. Создадим объект `Figure` и добавим на него двумерную гистограмму с помощью метода `add_histogram2d` (на момент написания книги этот тип диаграммы не поддерживался в Plotly Express). Мы просто выберем любые два индикатора, которые хотим проанализировать совместно, и передадим их параметрам `x` и `y`:

```
fig = go.Figure()
fig.add_histogram2d(x=df['Income share held by fourth 20%'],
                    y=df['GINI index (World Bank estimate)'],
                    colorscale='cividis')
```

3. Добавим заголовки для осей *x* и *y*:

```
fig.layout.xaxis.title = 'Income share held by fourth 20%'
fig.layout.yaxis.title = 'GINI index (World Bank estimate)'
fig.show()
```

Запуск этого кода приведет к выводу двумерной гистограммы, показанной на рис. 8.12.

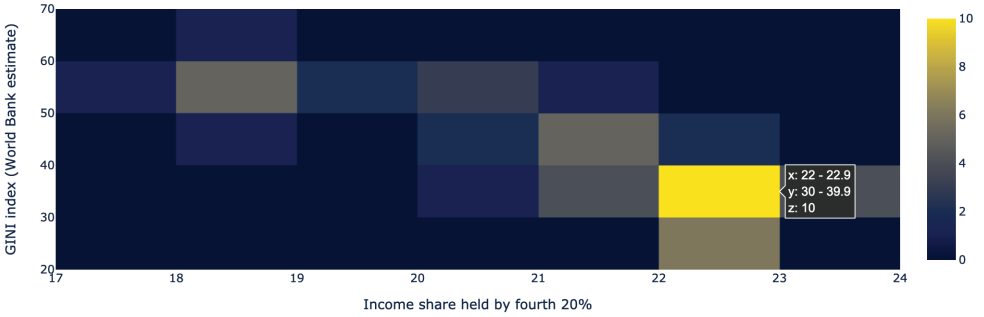


Рис. 8.12. Двумерная гистограмма

Частота значений на этой диаграмме выражается в несколько ином виде. Если на классической гистограмме за этот показатель отвечает высота столбиков, то на двумерной – непрерывная цветовая шкала. На графике, показанном на рис. 8.12, количество значений изменяется в диапазоне от 0 до 10, и больше всего значений показателя по оси *x* находится в интервале (22, 22.9), а по оси *y* – в интервале (30, 39.9). Частота, или значение, *z* здесь равна 10. В данном случае *z* используется по аналогии с третьим измерением, и этот показатель можно воспринимать как высоту прямоугольника.

Заметьте, что такой способ отображения сильно отличается от точечной диаграммы с двумя переменными. Там нас интересовала корреляция между двумя переменными и степень их изменчивости. Здесь же мы исследуем частоту встречаемых значений по двум разным переменным и ориентируемся на пересечения столбиков.

Мы рассказали далеко не про все возможности и опции частотного анализа значений при помощи гистограмм, но это просто невозможно физически. Далее мы перейдем к знакомству с еще одним интерактивным компонентом из состава Dash – `DataTable`.

Создание DataTable

Чисто технически `dash_table` является обособленным пакетом, который можно установить отдельно. Но обычно это делать не требуется, поскольку актуальная версия пакета устанавливается вместе с Dash.

Зачастую нам требуется вывести на дашборде табличную информацию в интерактивном виде. Кроме того, иногда хочется предоставить пользователям возможность обратиться к исходным данным для проведения собственного анализа. Компонент `DataTable` идеально подходит для таких ситуаций, к тому же он содержит богатые возможности для настройки в отношении цвета, шрифтов, их размеров и т. д. Таким образом, этот компонент обеспечивает нам еще один удобный способ визуализации данных. Мы познакомимся с некоторыми его возможностями, но, конечно, далеко не всеми.

Давайте создадим простую таблицу данных `DataTable` на основе датафрейма.

1. Отфильтруем датафрейм `poverty`, чтобы в нем остались только страны за 2000 год, и оставим в нем лишь колонки с названием страны и верхним и нижним сегментами в отношении дохода. Для этого воспользуемся методом `filter` с применением регулярных выражений:

```
df = poverty[poverty['year'].eq(2000)&poverty['is_country']].
filter(regex='Country Name|Income share.*10')
```

2. Создадим приложение JupyterLab с атрибутом `layout`:

```
app = JupyterDash(__name__, external_stylesheets=[dbc.themes.COSMO])
app.layout = html.Div([])
```

3. Теперь нам нужно разместить объект `DataTable` в элементе `div`. Для создания этого объекта требуется передать как минимум исходные данные и колонки для извлечения. Простейший способ сделать это – преобразовать датафрейм в словарь при помощи метода `to_dict()`. Параметр `columns` принимает на вход список словарей, в котором каждый словарь содержит ключи `name` и `id`. В первом хранится имя столбца, которое увидит пользователь, а во втором – идентификатор, который мы будем использовать:

```
DataTable(data=df.to_dict('records'),
          columns=[{'name': col, 'id': col}
                   for col in df.columns])
```

Запуск приложения при помощи метода `app.run_server()` приведет к выводу на экран таблицы, показанной на рис. 8.13.

Country Name	Income share held by highest 10%	Income share held by lowest 10%
Afghanistan		
Albania		
Algeria		
Angola	40.2	1
Argentina	37.7	0.9
Armenia		
Australia		
Austria	22.7	3.4
Azerbaijan		
Bangladesh	27.9	3.7
Belarus	24	3.1
Belgium	28.3	3.3
Belize		

Рис. 8.13. Таблица данных

Зачастую заголовки колонок в ваших таблицах данных не будут умещаться в отведенные им ячейки, как в нашем примере. Давайте посмотрим, как с этим можно справиться.

Настройка отображения таблицы данных (ширина и высота ячеек, отображение текста и т. д.)

Существует масса вариантов настроить отображение таблиц данных, и вы всегда можете обратиться к документации за этими способами. Сложнее бывает при изменении комбинации опций. В этом случае настройки могут влиять друг на друга, в результате чего таблица может отображаться не так, как вам бы того хотелось. Таким образом, всегда лучше изолировать разные настройки при отладке.

На рис. 8.13 мы отобразили только три столбца и первые несколько строк в таблице. Посмотрим, как можно вывести больше столбцов и позволить пользователю смотреть больше строк.

1. Модифицируем датафрейм `poverty`, оставив в нем только страны за 2000 год и колонки с названием страны и словами `Income share` в заголовке:

```
df = poverty[poverty['year'].eq(2000)&poverty['is_country']].
filter(regex='Country Name|Income share')
```

2. Поместим пустой компонент `DataTable` в `dbc.Col` с желаемой шириной, равной семи. Таблица автоматически примет ширину контейнера, в ко-

тором размещается, так что ее ширина в этом случае будет задана неявным образом:

```
    dbc.Col([
        DataTable()
    ], lg =7)
```

3. Теперь нужно определиться с тем, как будут себя вести заголовки столбцов, особенно если их длина будет превышать ширину колонки. Этот аспект можно настроить при помощи параметра `style_header`. Заметьте, что для заголовков, ячеек и таблицы имеется целый ряд параметров, начинающихся на `style_`, кроме того, у них есть и вариации с окончанием `_conditional`, например `style_cell_conditional`, для условной установки стилей. Мы определим стиль для заголовков с опцией `whiteSpace`, чтобы текст при необходимости мог переноситься на новую строку:

```
  style_header={'whiteSpace': 'normal'}
```

4. Также нам бы хотелось, чтобы во время вертикальной прокрутки заголовки находились на своем месте. Этого можно добиться при помощи параметра `fixed_rows`, как показано ниже:

```
  fixed_rows={'headers': True}
```

5. Чтобы следить за общей высотой таблицы, можно задать атрибут `height` в параметре `style_table` следующим образом:

```
  style_table={'height': '400px'}
```

6. Если в вашей таблице тысячи строк, она может оказаться слишком тяжелой для отображения на странице. Для снижения этого эффекта можно воспользоваться параметром `virtualization`. У нас таблица небольшая, но для примера мы добавим этот параметр:

```
  virtualization=True
```

Ниже показан весь код для создания таблицы, собранный воедино:

```
dbc.Col([
    DataTable(data=df.to_dict('records'),
              columns=[{'name': col, 'id': col}
                       for col in df.columns],
              style_header={'whiteSpace': 'normal'},
              fixed_rows={'headers': True},
              virtualization=True,
              style_table={'height': '400px'})
], lg =7),
```

Внешний вид получившейся таблицы данных показан на рис. 8.14.

Country Name	Income share held by fourth 20%	Income share held by highest 10%	Income share held by highest 20%	Income share held by lowest 10%	Income share held by lowest 20%	Income share held by second 20%	Income share held by third 20%
Cyprus							
Czech Republic							
Denmark	22.6	20.1	34.2	4.4	10.4	14.7	18.1
Djibouti							
Dominican Republic	20.1	40.2	56.2	1.2	3.7	7.7	12.3
Ecuador	18.1	45.9	60.7	0.9	3	7	11.2
Egypt, Arab Rep.							
El Salvador	20.5	39	55.8	1	3.2	7.8	12.7
Equatorial Guinea							
Eritrea							
Estonia							
Eswatini	17.5	44.1	59.5	1.8	4.5	7.5	10.9

Рис. 8.14. Таблица данных с настроенными опциями ширины, высоты, прокрутки и виртуализации

Полоса прокрутки будет показываться только при наведении мышью на эту область. На рисунке мы сохранили ее, чтобы продемонстрировать ее внешний вид. С помощью полосы прокрутки пользователь может сам перемещаться по таблице и искать нужные ему строки. Давайте посмотрим, как можно добавить таблице данных интерактивности и внедрить ее в приложение. Попутно мы рассмотрим еще несколько важных опций применительно к компоненту `DataTable`.

Добавление гистограмм и таблиц данных в приложение

Итак, мы готовы добавить таблицу данных в наше приложение и в функцию обратного вызова, которую мы уже написали. Давайте в табличном виде отобразим данные, использующиеся для построения гистограммы. Поскольку на гистограммах не отображаются сами точки данных (а только агрегаты), пользователю может быть полезно взглянуть на них в табличном виде.

Давайте дадим ему такую возможность.

1. Добавим новый элемент `div` непосредственно под графиком с гистограммой:

```
html.Div(id='table_histogram_output')
```

2. Добавим новый элемент вывода в объявление функции обратного вызова:

```
@app.callback(Output('indicator_year_histogram', 'figure'),
              Output('table_histogram_output', 'children'),
              Input('hist_multi_year_selector', 'value'),
              Input('hist_indicator_dropdown', 'value'),
              Input('hist_bins_slider', 'value'))
```

3. Теперь добавим при создании объекта `DataTable` несколько параметров, разместив их после параметра `style_table={'height': '400px'}`. Начнем с возможности сортировать столбцы с помощью параметра `sort_action`:

```
sort_action='native'
```

4. Теперь добавим возможность выполнять фильтрацию данных в колонках посредством передачи параметра `filter_action`. Это приведет к появлению ячеек под заголовками колонок таблицы, в которых пользователь сможет ввести значение для фильтра и нажать на кнопку **Enter**, после чего в таблице останутся только строки, соответствующие введенному шаблону:

```
filter_action='native'
```

5. Добавим возможность экспортировать данные из таблицы в файл CSV, передав параметр `export_format`:

```
export_format='csv'
```

6. Установим минимальную ширину для столбцов при помощи опции `minWidth` в параметре `style_cell`, чтобы избежать проблем с форматированием из-за различий в заголовках столбцов:

```
style_cell={'minWidth': '150px'}
```

7. Наконец, добавим переменную с таблицей в инструкцию `return` в конце функции, чтобы она возвращала два значения вместо одного:

```
return fig, table
```

В результате под нашей гистограммой в приложении появится таблица с исходными данными для нее, с которой пользователь сможет интерактивно взаимодействовать. На рис. 8.15 показан внешний вид этой секции приложения.

Рис. 8.15. Таблица с исходными данными для гистограммы

Как видите, слева вверху над таблицей появилась кнопка **Export**, с помощью которой можно загрузить данные в формате CSV. К тому же слева от заголовков колонок отображаются стрелки, позволяющие сортировать данные по возрастанию и убыванию, а под заголовками – поля для осуществления фильтрации.

Чтобы добавить созданный функционал в наше приложение, необходимо скопировать новые компоненты туда, где вы хотите их видеть. Возможно, лучшее место для них найдется непосредственно под картой.

Для внедрения интерактивности перенесите созданную функцию обратного вызова и вставьте ее после объявления макета.

Все эти действия мы уже проделывали не раз, так что вы прекрасно справитесь с этим самостоятельно.

После всех внесенных изменений наше приложение значительно разрослось. В верхней его части у нас находятся две исследовательские секции. Географическая карта позволяет пользователю выбрать метрику и проанализировать ее визуально по странам. Также он может выбрать год и/или просматривать аналитику с помощью меняющихся слайдов, как на видео. К тому же выбор индикатора приведет к выводу полной информации о нем, включая возможные ограничения. Ниже пользователь может выбрать показатель и один или несколько лет для анализа распределения значений метрики при помощи гистограмм. При этом он волен сам выбирать, какое количество столбиков отображать. Под гистограммами пользователь увидит таблицу с исходными данными, с которой сможет интерактивно взаимодействовать и даже выгрузить в формате CSV.

После завершения исследования пользователь может обратиться к трем специализированным диаграммам для дальнейшего анализа выбранных индикаторов.

Поздравляем! Мы завершили вторую часть книги, так что сейчас самое время подвести итоги этой главы и части в целом, а также подготовиться к чтению заключительной части.

Заключение

Начали мы эту главу с перечисления различий между гистограммой и другими типами визуализаций, о которых говорили ранее. Мы увидели, как легко можно создать и настроить гистограмму, после чего поговорили об интерактивности гистограмм и их связи с другими компонентами посредством функций обратного вызова.

Далее мы рассмотрели довольно редко встречающуюся разновидность гистограмм, называемую двумерной гистограммой, которая позволяет проводить совместный частотный анализ сразу по двум переменным.

После этого мы познакомились с компонентом `DataTable`, позволяющим визуализировать данные в табличном виде. Функционал этого компонента довольно богат, но мы рассмотрели только самые базовые возможности. В нашем приложении мы использовали этот компонент для отображения исходных данных для построения гистограммы с возможностью интерактивного взаимодействия с ними. Попутно мы узнали, как можно настраивать внешний вид этих таблиц.

В конце главы мы добавили таблице данных интерактивности при помощи написанной ранее функции обратного вызова и внедрили новый компонент в наше приложение.

Теперь давайте кратко пройдемся по тому, что мы уже успели изучить в этой книге, и подготовимся к заключительной части.

Что мы узнали из первых двух частей книги

В первой части книги мы познакомились с концепцией и основами фреймворка Dash. Мы узнали о структуре фреймворка и управлении визуальными элементами, входящими в его состав. После этого освоили основы интерактивности в Dash, реализуемые с помощью функций обратного вызова. Это позволило нам создать свое первое полностью интерактивное приложение. Следом мы познакомились со структурой объекта `Figure` и научились манипулировать им с целью построения нужных нам диаграмм. Затем погрузились в мир преобразования данных и узнали, насколько это важный процесс для дальнейшей визуализации. Мы научились видоизменять структуру наших данных, приводя их к виду, удобному для дальнейшего анализа, что поспособствовало изучению модуля Plotly Express, помогающего легко и интуитивно визуализировать данные.

Во второй части книги мы главным образом сосредоточили свое внимание на различных типах визуализации и использовании применительно к ним интерактивного подхода к взаимодействию. Мы подкрепили на практике все знания, полученные в первой части книги. Вместе с тем мы постепенно расширяли наше приложение, добавляя в него новые компоненты и функционал. При этом нам необходимо было заботиться о том, чтобы не нарушить целостность приложения и не сломать то, что уже было сделано ранее. Это должно было научить вас интеграционному подходу к проектированию приложения в отношении изменения и дополнения его функционала. Да, мы рассмотрели далеко не все имеющиеся компоненты и диаграммы, но основные принципы их использования не меняются, и вы можете без труда самостоятельно восполнить эти пробелы.

В третьей части книги мы поговорим о более общих концепциях, применяемых в отношении приложений, таких как ссылки, дополнительные опции функций обратного вызова и процесс развертывания приложений. Но в следующей главе мы уделим некоторое внимание приемам машинного обучения. Наш набор данных содержит множество стран, лет и индикаторов, и проанализировать все их возможные комбинации было бы очень затруднительно. Таким образом, мы познакомимся с некоторыми техниками, помогающими отследить ключевые тенденции и зависимости в наших исходных данных.

Часть III

Развитие приложений. Новый уровень

В этой части мы продолжим улучшать и развивать наши приложения, а также рассмотрим несколько новых техник и стратегий и научимся развертывать приложение.

Содержание этой части:

- глава 9 *«Машинное обучение: пусть данные говорят сами за себя»;*
- глава 10 *«Ускорение работы приложений с помощью улучшений функций обратного вызова»;*
- глава 11 *«Ссылки и многостраничные приложения»;*
- глава 12 *«Развертывание приложения»;*
- глава 13 *«Следующие шаги».*

Глава 9

Машинное обучение: пусть данные говорят сами за себя

В предыдущей главе мы на примере гистограмм познакомились с техникой визуализации агрегатов, а не самих исходных данных. Иными словами, мы анализировали данные о данных. В этой главе мы сделаем еще несколько шагов в этом направлении и при помощи техники *машинного обучения* (machine learning) продемонстрируем приемы категоризации и кластеризации данных. Как вы узнаете из этой главы, существует множество вариантов и их комбинаций, которые можно исследовать. И в этом нам помогут интерактивные дашборды. Согласитесь, для пользователя было бы очень утомительно, если бы ему пришлось для исследования каждого аспекта вручную строить для него отдельный график.

Эту главу нельзя рассматривать как введение в машинное обучение, но в то же время она не предполагает наличия у читателя никаких особых знаний в этой области. Мы будем применять технику кластеризации по методу *k*-средних (KMeans clustering), используя для этого библиотеку `sklearn`. В процессе работы будем разбивать наблюдения на кластеры по схожим признакам, общим в рамках одной группы и отличающимся от других. В результате построим очень простую модель на основе одномерного набора данных и посмотрим, как ее можно применить для выполнения кластеризации стран в нашем датафрейме `poverty`.

Если вы знакомы с принципами машинного обучения, возможно, чтение этой главы даст вам новые идеи относительно того, как пользователи могут самостоятельно использовать построенные модели в процессе анализа данных. Если нет – после прочтения вы наверняка зададитесь целью узнать побольше об этой перспективной области знаний.

Темы, которые будут рассмотрены в главе:

- кластеризация данных;
- поиск оптимального количества кластеров;
- кластеризация стран по численности населения;
- подготовка данных с использованием библиотеки scikit-learn;
- создание интерактивного приложения с применением кластеризации по методу *k*-средних.

Технические требования

В этой главе мы будем активно использовать пакеты `sklearn` и `NumPy`. Также мы продолжим работать с библиотеками, с которыми познакомились ранее. Для преобразования данных мы будем использовать пакет pandas, а библиотека JupyterLab поможет нам опробовать наши решения в действии. Кроме того, мы продолжим использовать пакеты Dash, JupyterDash, Dash Core Component, Dash HTML Components, Dash Bootstrap Components, Plotly и Plotly Express. Пакетом `sklearn` мы воспользуемся для построения моделей машинного обучения и подготовки данных.

Исходный код к этой главе располагается в хранилище GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_09**.

Сопроводительные видеофрагменты к главе можно посмотреть по адресу **https://bit.ly/3x8PAmt**.

Кластеризация данных

Что такое *кластеризация* (clustering) и для чего она применяется? Давайте начнем с элементарного примера. Представьте, что у нас есть группа людей, для которых нам необходимо пошить футболки. При этом у нас есть одно производственное ограничение, состоящее в том, что мы можем шить футболки только одного размера. Список размеров выглядит следующим образом: [1, 2, 3, 4, 5, 7, 9, 11]. Подумайте, какой подход к решению этой задачи вы бы выбрали. Мы будем решать ее с использованием техники *кластеризации по методу k-средних* (KMeans clustering). Так давайте приступим.

1. Импортируем все необходимые пакеты и модели. `NumPy`, который мы уже упоминали ранее, мы загрузим как библиотеку, а из пакета `sklearn` импортируем единственную модель, с которой будем работать, как показано ниже:

```
import numpy as np
from sklearn.cluster import KMeans
```

2. Создадим набор данных о размерах в требуемом формате. Обратите внимание, что каждое наблюдение (размер одежды человека) должно быть представлено в виде списка, так что мы применим метод `reshape` к массиву `NumPy`, чтобы этого добиться:

```
sizes = np.array([1, 2, 3, 4, 5, 7, 9, 11]).reshape(-1, 1)
sizes
array([[ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
```

```
       [ 7],
       [ 9],
       [11]])
```

> **Примечание**
>
> Жирным шрифтом показан вывод программы, его вводить с клавиатуры не требуется.

3. Создадим экземпляр модели `KMeans` с требуемым количеством кластеров. Важной особенностью этой модели является то, что мы предоставляем ей требуемое количество кластеров, на которые нужно разбить набор. В данном случае у нас есть очень жесткое ограничение в виде того, что мы можем себе позволить пошить футболки только одного размера. Таким образом, наша задача сводится к тому, чтобы найти центр нашего единственного кластера. Позже мы исследуем предпосылки и последствия выбора количества кластеров. Сейчас же запустим приведенный ниже код:

```
kmeans1 = KMeans(n_clusters=1)
```

4. Теперь нам нужно обучить модель с использованием метода `fit`. Это означает, что мы хотим, чтобы созданная нами модель обучилась на нашем наборе данных, следуя конкретному алгоритму и учитывая переданные ей параметры. Обучение модели выполняется следующим образом:

```
kmeans1.fit(sizes)
KMeans(n_clusters=1)
```

Итак, мы получили модель, обученную на нашем наборе данных о размерах футболок. Пришло время посмотреть на ее атрибуты. По общему соглашению имена результирующих атрибутов обученной модели имеют на конце символ подчеркивания (_), что мы здесь и увидим. Мы можем запросить информацию о центрах кластеров, на которые был разбит наш набор данных, при помощи атрибута `cluster_centers_`. В данном случае, поскольку кластер у нас всего один, его центр будет располагаться в точке, соответствующей среднему значению чисел в нашем наборе данных. Давайте в этом убедимся:

```
kmeans1.cluster_centers_
array([[5.25]])
```

Мы получили результирующий атрибут в виде списка списков. Очевидно, что центр нашего кластера лежит на отметке `5.25`. Вам справедливо может показаться, что это какой-то уж слишком извращенный способ получения среднего значения чисел. Достаточно сравнить полученный результат со средним арифметическим нашего набора чисел:

```
sizes.mean()
5.25
```

И правда, центр нашего кластера в точности совпадает со средним значением исходных чисел, и мы именно эту информацию и запросили. На рис. 9.1 вы можете увидеть наши точки данных и центр образованного кластера графически.

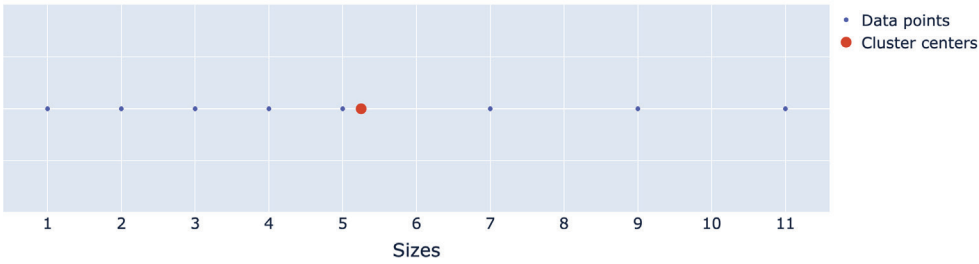


Рис. 9.1. Точки данных и центр кластера

Это очень простой график, на котором мы вывели все наши точки на одной оси *x* с фиксированной произвольной координатой по оси *y*.

Оценить качество или эффективность обучения построенной нами модели можно разными способами, и один из них связан с оценкой атрибута `inertia_`. Этот атрибут принадлежит модели и может быть извлечен после ее обучения с использованием точечной нотации, как показано ниже:

```
kmeans1.inertia_
85.5
```

Атрибут `inertia_` выражается как сумма квадратов расстояний исходных точек до ближайших к ним центров кластеров. Если модель обучена качественно, расстояния до центров кластеров должны стремиться к нулю, что будет означать их максимальную приближенность к центрам соответствующих кластеров. Идеальная модель характеризуется нулевым показателем `inertia_`. А теперь взгляните на это под другим углом. Как вы можете догадаться, попытка разбить данные на один кластер приведет к созданию худшей из возможных моделей, поскольку центр кластера у нас будет только один, а значит, и расстояния от него до наиболее удаленных точек будут очень большими.

Соответственно, улучшить модель мы можем, добавив в наши исходные требования большее количество кластеров, на которые нужно разбивать данные. Очевидно, что это позволит снизить расстояния от точек данных до центров кластеров, которых будет уже несколько.

Теперь представьте, что я принес вам хорошую весть. Нам выделили дополнительный бюджет, что дало нам возможность шить футболки двух размеров. В переводе на язык машинного обучения это означает, что нам нужно создать новую модель с двумя кластерами. Повторим наши предыдущие шаги, изменив значение параметра `n_clusters` на 2, как показано ниже:

```
kmeans2 = KMeans(n_clusters=2)
kmeans2.fit(sizes)
kmeans2.cluster_centers_
```

```
array([[3.],
       [9.]])
```

Теперь у нас есть два центра кластеров, что видно на выводе.

Но нам недостаточно знать только центры кластеров, мы хотим понимать, какому кластеру принадлежит каждая точка в нашем наборе данных, или, иными словами, футболку какого размера мы дадим каждому из участников нашего эксперимента. Также мы должны посчитать, сколько и каких размеров нам нужно сшить футболок.

Требуемую нам информацию содержит атрибут `labels_`:

```
kmeans2.labels_
array([0, 0, 0, 0, 0, 1, 1, 1], dtype=int32)
```

Обратите внимание, что метки были выданы начиная с нуля, а не с единицы. Также заметьте, что сами числа меток ровным счетом ничего не значат. Точки с нулевыми метками не обязательно принадлежат первому кластеру, а точки с единичными метками ни в чем не превосходят коллег из соседней группы. Это просто метки – как если бы мы назвали группы *А* и *Б*.

Мы можем сопоставить метки и наши значения, воспользовавшись функцией `zip`, как показано ниже:

```
list(zip(sizes, kmeans2.labels_))
[(array([1]), 0),
 (array([2]), 0),
 (array([3]), 0),
 (array([4]), 0),
 (array([5]), 0),
 (array([7]), 1),
 (array([9]), 1),
 (array([11]), 1)]
```

Этот шаг обретет особую значимость позже, когда мы будем использовать данные метки на графиках.

Давайте визуализируем два центра кластеров на фоне наших точек данных. На рис. 9.2 показано, где именно располагаются эти центры относительно наших исходных точек.

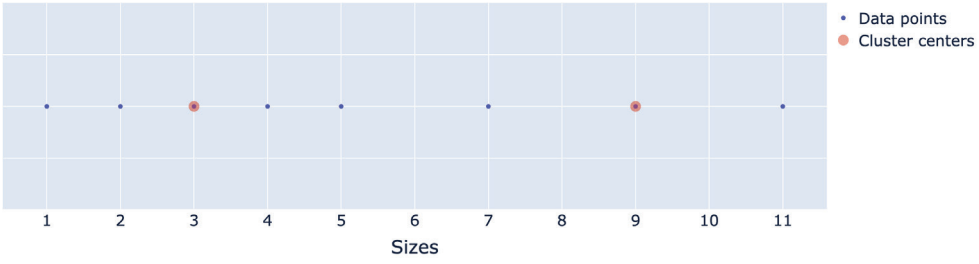


Рис. 9.2. Исходные точки данных и центры кластеров

На графике все сразу становится понятно. Мы видим невооруженным взглядом, что первые пять точек располагаются ближе к центру левого кластера, а последние три – к центру правого. И в этом свете центры кластеров, располагающиеся на отметках 3 и 9, выглядят вполне логично – каждый из них представляет собой среднее арифметическое значений входящих в кластер точек. Давайте убедимся, что, увеличив количество кластеров до двух, мы повысили качество нашей модели. Для этого снова обратимся к атрибуту `inertia_`:

```
kmeans2.inertia_
18.0
```

Неплохое такое улучшение – сразу с 85,5 до 18,0! Но здесь нет ничего удивительного. Каждое увеличение количества кластеров будет приводить к улучшению модели, пока не будет достигнут идеальный показатель, равный нулю. Но как нам найти нужное количество кластеров? Этим мы сейчас и займемся.

Поиск оптимального количества кластеров

В этом разделе мы посмотрим, какие у нас есть инструменты для нахождения оптимального количества кластеров, но для начала давайте взглянем на рис. 9.3, чтобы понять, как меняется положение центров кластеров с изменением их количества от одного до восьми.

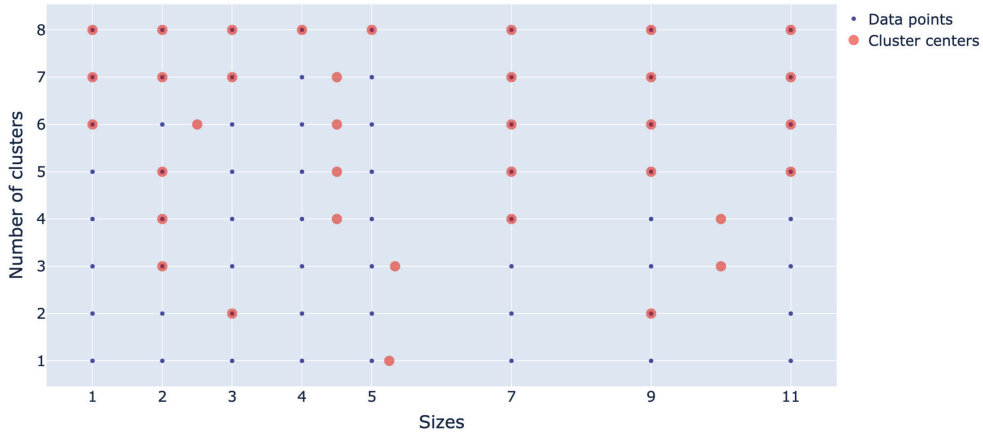


Рис. 9.3. Точки данных и центры кластеров для всех возможных случаев

Мы видим, как перемещаются центры кластеров по отношению к исходным точкам данных. В результате, когда количество кластеров сравняется с количеством точек, мы получим теоретически идеальное качество модели – каждая точка и будет являться центром своего кластера, в который будет входить она одна.

В реальном мире доводить модель до такого «идеала» нет смысла по двум причинам. Во-первых, это может быть губительно с точки зрения затрат. Представьте, что вам нужно сшить 1000 футболок не двух размеров, а нескольких сотен. Во-вторых, в практических задачах чаще всего не наблюдается значи-

тельного улучшения качества модели после достижения определенного предела. Представьте, что у нас есть два человека в группе с размерами 5,3 и 5,27. Очевидно, что они вполне могут носить футболки одного размера.

Итак, мы определились с тем, что оптимальное количество кластеров лежит где-то в интервале от единицы до количества уникальных точек в наборе данных. И теперь нам необходимо точно вычислить это значение. Один из способов состоит в определении ценности каждого нового кластера. Происходит ли существенное повышение качества модели при добавлении очередного кластера? Графическое выражение этого способа получило название *метод локтя* (elbow technique). Этот метод состоит в выводе на графике значений качества модели для всех вариантов количества кластеров и определении точки перелома. Давайте проделаем эту операцию.

Пройдемся в цикле от одного до восьми включительно и будем на каждой итерации создавать модель с заданным количеством кластеров, вычислять ее качество и записывать этот показатель в список. Этот простой скрипт показан ниже:

```
inertia = []
for i in range(1, 9):
    kmeans = KMeans(i)
    kmeans.fit(sizes)
    inertia.append(kmeans.inertia_)
inertia

[85.5, 18.0, 10.5, 4.5, 2.5, 1.0, 0.5, 0.0]
```

Как и ожидалось, качество модели в конечном счете достигло идеала. Теперь выведем полученные результаты на графике для иллюстрации метода локтя:

```
import plotly.graph_objects as go
fig = go.Figure()
fig.add_scatter(x=list(range(1, 9)), y=inertia)
fig.layout.xaxis.title = 'Number of clusters'
fig.layout.yaxis.title = 'Inertia'
fig.show()
```

На рис. 9.4 показан график, который получился в итоге.

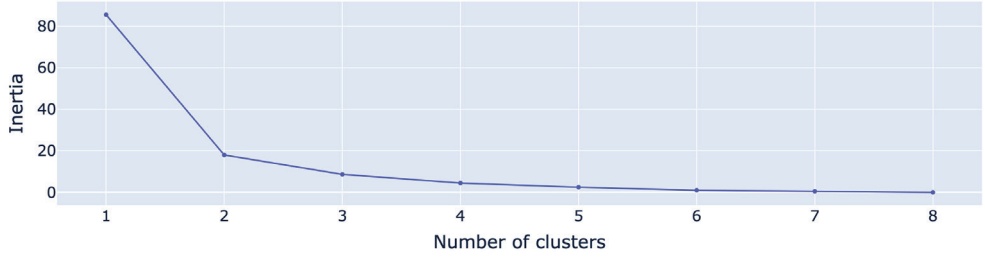


Рис. 9.4. Качество модели для разного количества кластеров

Как видите, максимальное увеличение качества модели произошло при переходе от бессмысленного количества кластеров, равного единице, к двум кластерам. Дальше улучшение происходит не столь стремительными темпами. Таким образом, примерно на отметке в 3–4 кластера прирост качества почти полностью сглаживается, а значит, в этом районе и находится оптимальное количество кластеров для нашего ряда данных. К тому же здесь нам всю картину портит отметка с одним кластером, поскольку мы знаем, что этот вариант нам точно не подходит. На рис. 9.5 показан тот же график, но с исключенной левой точкой данных.

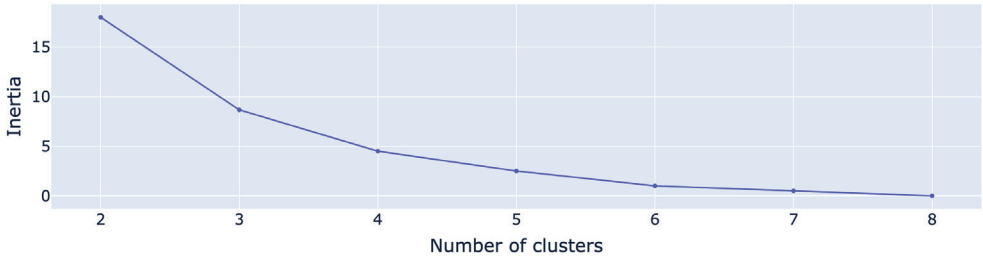


Рис. 9.5. Исключение варианта с единственным кластером

Здесь картина существенно изменилась, и мы видим, что без дополнительной информации нам будет трудно принять решение об оптимальном количестве кластеров.

Мы рассмотрели наиболее простой пример одномерных данных. Но метод *k*-средних и машинное обучение в целом зачастую выходят за рамки одного измерения и решают более объемные задачи. При этом концепция остается прежней: мы пытаемся найти такие центры кластеров и такое их количество, чтобы расстояние между ними и точками данных было минимальным. На рис. 9.6 показан пример кластеризации двумерных данных.

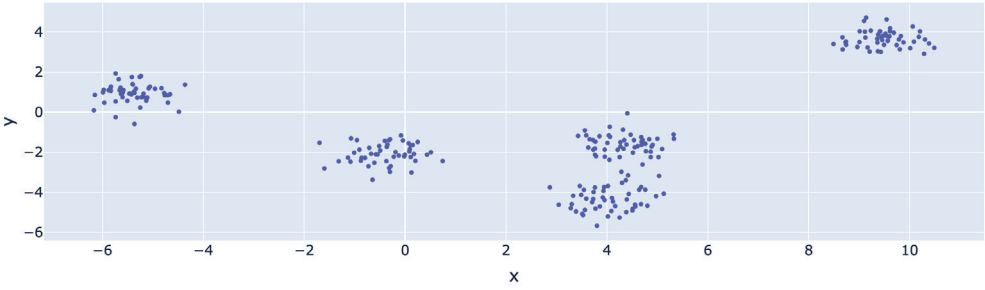


Рис. 9.6. Кластеризованные точки данных в двух измерениях

Мы могли бы добавить дополнительное измерение в наш ряд данных. К примеру, по оси *x* мы бы откладывали рост людей, а по оси *y* – их вес. Что нам на это скажет модель? Конечно, в реальной жизни данные редко так явно разделены на группы, как показано на рис. 9.6. Вы также должны понимать, что неверно выбранное количество кластеров может привести к значительной потере точности. Например, если исходя из того же рис. 9.6 предложить разделить данные

на три кластера, очевидно, что средние три группы точек будут объединены в один кластер, хотя мы видим, что в них есть серьезные различия. В то же время если разделить этот набор данных на семь или восемь кластеров, мы получим ненужные деления внутри одного логического кластера. На графике это будет означать, что мы перешагнули далеко за точку перелома, в которой находится оптимальное количество кластеров.

Теперь давайте попробуем реализовать принципы кластеризации применительно к нашему рабочему набору данных.

Кластеризация стран по численности населения

Для начала попробуем оперировать одним индикатором, с которым уже хорошо знакомы, а именно с показателем численности населения. После этого добавим нашему решению интерактивности. В целом же мы будем выполнять кластеризацию стран по количеству проживающих в них людей.

Начнем сразу с практики. Представьте, что вас попросили условно разделить страны на карте мира по численности населения. Групп при этом должно быть только две: страны с высокой численностью населения и низкой. Как вы это будете делать? Где провести красную линию и какое значение может рассматриваться в качестве критерия? А если позже вас попросят увеличить количество кластеров до трех или четырех? Как вы измените свои расчеты?

Давайте применим кластеризацию по методу *k*-средних с одним измерением, после чего установим нужные соответствия с исходными данными.

1. Импортируем пакет `pandas` и создадим датафрейм `poverty`:

```
import pandas as pd
poverty = pd.read_csv('data/poverty.csv')
```

2. Объявим переменные для выбранного года и индикатора:

```
year = 2018
indicators = ['Population, total']
```

3. Создадим объект `KMeans` с желаемым количеством кластеров:

```
kmeans = KMeans(n_clusters=2)
```

4. Ограничим наш датафрейм только по странам и выбранному году следующим образом:

```
df = poverty[poverty['year'].eq(year) & poverty['is_country']]
```

5. Создадим объект `data` со списком выбранных столбцов (в данном случае мы выбрали единственный столбец). Обратите внимание, что мы обращаемся к атрибуту `values`, который возвращает массив `NumPy`:

```
data = df[indicators].values
```

6. Обучим модель на наших данных:

```
kmeans.fit(data)
```

Итак, мы выполнили обучение модели и готовы визуализировать полученные данные. Если помните, в главе 7 мы упоминали, что для создания визуализации в виде графической карты нам нужно, чтобы в датафрейме присутствовал столбец с названиями или кодами стран. Этого достаточно, чтобы отобразить их на карте. Если мы хотим раскрасить страны в различные цвета, понадобится еще один столбец (или любой списочный объект) с соответствующими значениями.

В обученном нами объекте `kmeans` содержатся метки, говорящие о принадлежности стран тому или иному кластеру. Мы используем эти метки для цветового обозначения стран на карте. Обратите внимание, что попутно мы изменим тип данных меток на строковый, чтобы Plotly Express воспринимал их как категориальные, а не непрерывные переменные. Код отображения данных на карте представлен ниже:

```
px.choropleth(df,
              locations='Country Name',
              locationmode='country names',
              color=[str(x) for x in kmeans.labels_])
```

Результат показан на рис. 9.7.

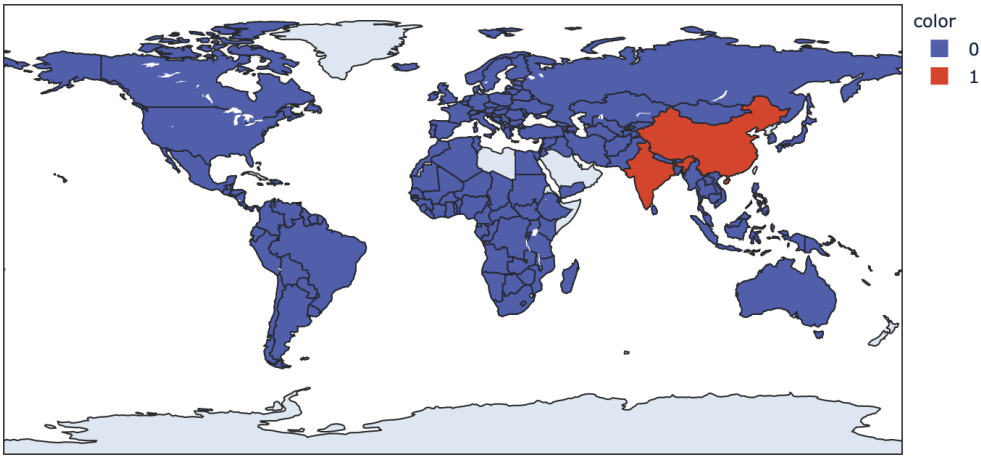


Рис. 9.7. Страны, разделенные по численности населения

Поскольку у нас уже есть рабочий шаблон из опций для карт, мы можем скопировать его и использовать для расширения визуализации и ее адаптации под наше приложение. Давайте посмотрим, как будет выглядеть разделение стран на карте с использованием одного, двух, трех и четырех кластеров, а затем обсудим увиденное. На рис. 9.8 показаны четыре карты с соответствующим количеством кластеров.

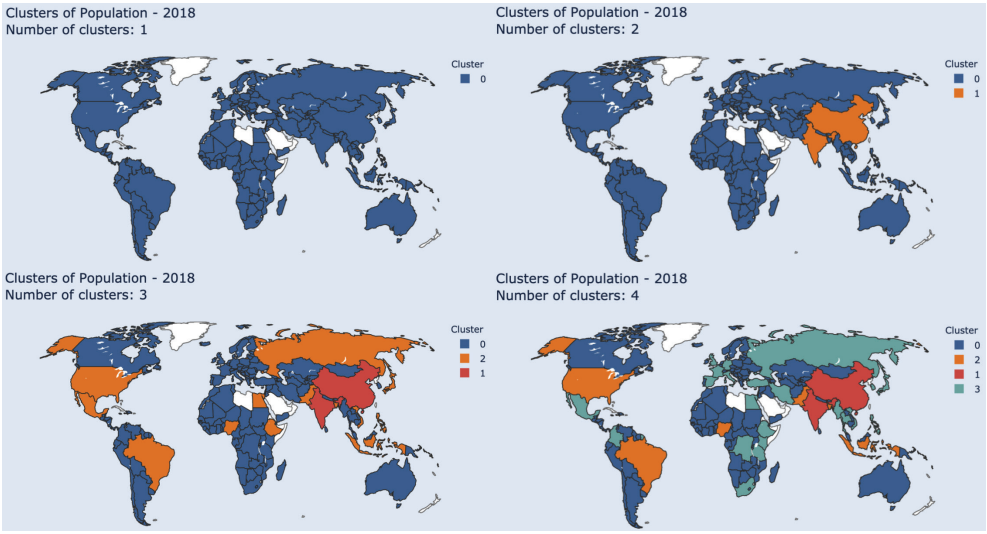


Рис. 9.8. Множественная кластеризация стран по численности населения

Вполне ожидаемо, что на карте с одним кластером все страны оказались окрашены в один цвет. Интереснее становится начиная с двух кластеров. Здесь в отдельный кластер были выделены две страны (Китай и Индия) с большой и очень схожей численностью населения: 1,39 и 1,35 млрд человек соответственно. В третьей стране по этому показателю (США) проживает «всего» 327 млн человек. Как видим, с помощью кластеризации мы выделили две страны в отдельный кластер, а все остальные попали в другой. Вряд ли этот метод можно считать оптимальным, но мы сами указали разбить страны на две группы.

При добавлении третьего кластера появилась дополнительная группа с умеренной численностью населения, включающая США, Россию, Бразилию, Мексику, Японию и некоторые другие страны. При разделении на четыре кластера Россия и Япония были определены в третью группу, хотя ранее они числились во второй.

На данный момент вы уже должны были ухватить основную мысль, и мы можем двигаться дальше. Давайте дадим пользователю возможность самому выбирать, какой индикатор анализировать и на какое количество кластеров разбивать выборку. Но для этого нам нужно провести определенные преобразования набора данных.

Подготовка данных с использованием библиотеки scikit-learn

Библиотека `scikit-learn` является одной из наиболее популярных в Python для работы с алгоритмами машинного обучения. Кроме того, она отлично адаптирована для работы с другими ключевыми пакетами в области анализа данных, такими как `NumPy`, `pandas` и `matplotlib`. Мы будем использовать ее для моделирования данных и их предварительной обработки.

На данный момент у нас есть две задачи, которые нам необходимо решить: пропущенные значения и масштабируемость данных. Давайте рассмотрим для каждой из них по два простых примера, после чего придем к решению применительно к нашему набору данных. Начнем с пропущенных значений.

Заполнение пропущенных значений

Моделям нужны данные, и они не знают, что им делать с пропущенными значениями. Решение о том, как именно обращаться с такими значениями в исходных данных, принимаете именно вы.

Здесь есть несколько подходов, а оптимальный вы сможете выбрать исходя из конкретной задачи и характера данных. Мы не будем вдаваться в тонкости выбора подхода, а для простоты демонстрации используем обобщенный алгоритм замены пропущенных значений на наиболее подходящие.

Давайте посмотрим на примере, как можно осуществить замену отсутствующих данных.

1. Создадим простейший набор данных с пропущенным значением в подходящем формате, как показано ниже:

```
data = np.array([1, 2, 1, 2, np.nan]).reshape(-1, 1)
```

2. Импортируем класс `SimpleImputer`:

```
from sklearn.impute import SimpleImputer
```

3. Создадим экземпляр загруженного класса со стратегией усреднения значений, принятой по умолчанию. Как вы можете догадаться, это далеко не единственная стратегия, доступная при замене пропущенных значений:

```
imp = SimpleImputer(strategy='mean')
```

4. Проведем обучение модели на данных. В этот момент происходит обучение с учетом заданных при создании объекта условиях:

```
imp.fit(data)
```

5. Преобразуем данные. Теперь, когда модель обучилась, можно провести преобразование согласно заданным правилам. Метод `transform` присутствует во множестве моделей и может иметь разные реализации в зависимости от контекста. В нашем случае преобразование будет означать подстановку пропущенных значений с использованием концепции *усреднения* (mean):

```
imp.transform(data)
array([[1. ],
       [2. ],
       [1. ],
       [2. ],
       [1.5]])
```

Как видите, модель преобразовала наши данные, заменив пропущенное значение на число 1,5. Если посмотреть на остальные значения в наборе (`[1, 2, 1, 2]`), легко можно понять, что число 1,5 было получено путем их усреднения. Мы могли бы также применить другую стратегию при заполнении пропущенных значений, включая вычисление медианы или моды (наиболее часто встречающегося значения). У каждой стратегии есть свои преимущества и недостатки.

Теперь поговорим о масштабировании данных.

Масштабирование данных при помощи scikit-learn

На рис. 9.6 мы видели, как может выглядеть процесс кластеризации данных в двух измерениях. Если бы мы хотели кластеризовать наш датафрейм `poverty` по двум столбцам, один из них был бы отображен на оси *x*, а второй – на оси *y*. А теперь представьте, что было бы, если бы на одной оси был показатель в абсолютных значениях (например, численность населения), а на второй – в процентах. В этом случае одни данные варьировались бы от 0 до 1,4 млрд, а другие – от 0 до 1 (или от 0 до 100, если выражать их в процентах). В результате любые отклонения процентного показателя не оказывали бы никакого значимого влияния на вычисляемые расстояния, а учитывалась бы только численность населения по странам в непропорциональном виде. Одно из решений состоит в *масштабировании* (scaling) данных.

Существуют разные способы масштабирования, мы же рассмотрим один из них, называемый *стандартным масштабированием* (standard scaling). Класс `StandardScaler` присваивает точкам данных *z-оценки*, или *стандартные оценки* (z-score), и нормализует их. *Z*-оценка вычисляется путем расчета разницы между значением и средним арифметическим по показателю и последующего деления результата на стандартное отклонение. Есть и другие методы расчета, но мы рассмотрим эту концепцию с целью упрощения примера.

1. Создадим простой набор данных:

```
data = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
```

2. Импортируем класс `StandardScaler` и создадим его экземпляр, как показано ниже:

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

3. Обучим модель на данных и преобразуем их. Для удобства во многих моделях, в которых присутствуют методы `fit` и `transform`, есть и объединенный метод `fit_transform`, которым мы сейчас и воспользуемся:

```
scaler.fit_transform(data)
array([[-1.41421356],
       [-0.70710678],
       [ 0.        ],
       [ 0.70710678],
       [ 1.41421356]])
```

Итак, мы выполнили *z*-преобразование наших данных. Обратите внимание, что среднее значение (3) было преобразовано в ноль, все значения больше трех стали положительными, а меньше – отрицательными. Сами числа характеризуют отклонение соответствующего значения от среднего.

Этот способ позволяет при наличии в наборе данных разных переменных нормализовывать их, сравнивать друг с другом и использовать совместно. В конце концов, нас интересует то, насколько велико значение и как далеко оно отстоит от среднего. К примеру, значение индекса Джини, равное 90, можно назвать экстремальным. Это примерно соответствует численности населения 1 млрд человек, если переводить в плоскость другой переменной. Если использовать эти переменные совместно, 1 млрд перевесит все другие цифры и исказит показатель. Стандартизация позволяет работать с разными данными в одном масштабе. Это не идеальный метод, но он помогает нивелировать изъяны, связанные с разными единицами измерения. Применяя этот способ, мы можем использовать при выполнении кластеризации более одной переменной.

Создание интерактивного приложения с применением кластеризации по методу *k*-средних

Теперь давайте соберем все воедино и создадим интерактивное приложение, использующее метод кластеризации. Мы позволим пользователю выбрать год и индикатор(ы) для анализа. Он сможет также указать количество кластеров, на которые следует разбить страны. В результате будет отображена картограмма с цветовой заливкой в зависимости от выбранного количества кластеров.

Учтите, что при выборе нескольких индикаторов вам может быть затруднительно сделать какие-либо выводы по результатам. Причина состоит в использовании сразу нескольких измерений. Кроме того, необходимо быть немного экономистом, чтобы понимать, какие индикаторы можно сочетать и какие выводы делать на основании полученного графика.

На рис. 9.9 показан внешний вид нашего будущего приложения.

Как видите, наше приложение будет обладать довольно широким спектром возможностей. Да, правильно интерпретировать полученные данные может быть непросто, но напомним, что мы просто проводим исследование доступных нам переменных с помощью одной техники и с использованием ограниченного набора опций.

На протяжении книги мы создали уже так много слайдеров и выпадающих списков, что не будем здесь подробно останавливаться на этом процессе. Мы пропишем только идентификаторы, а оставшуюся часть конструкторов вы сможете заполнить самостоятельно. Как видно на рис. 9.9, в нашем приложении будет два слайдера, один выпадающий список и один график, так что давайте создадим эти компоненты и дадим им идентификаторы. Как обычно, этот код должен присутствовать внутри элемента `app.layout`:

```
dcc.Slider(id='year_cluster_slider', ...),
dcc.Slider(id='ncluster_cluster_slider', ...),
dcc.Dropdown(id='cluster_indicator_dropdown', ...),
dcc.Graph(id='clustered_map_chart', ...)
```

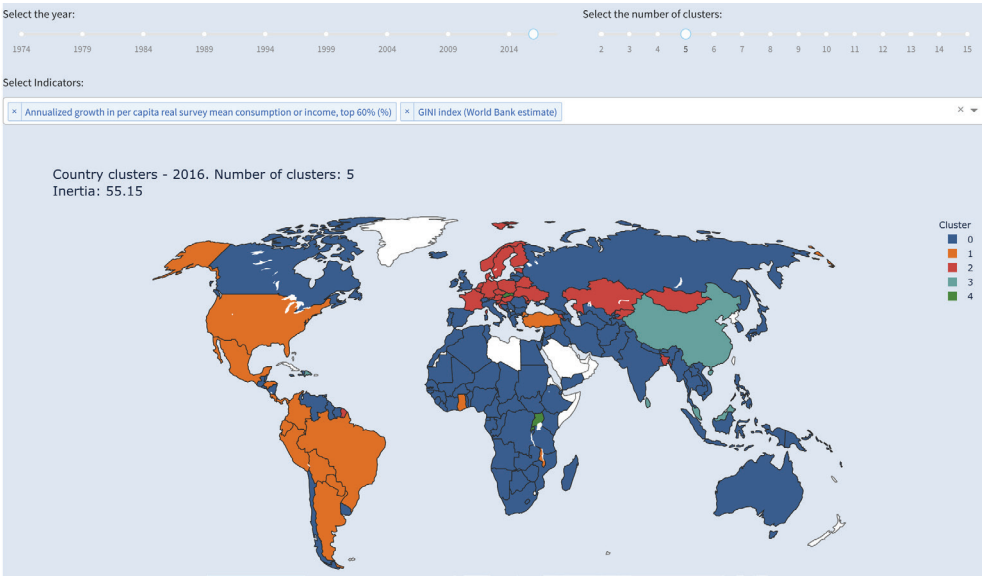


Рис. 9.9. Интерактивное приложение с кластеризацией по методу *k*-средних

Теперь создадим функцию обратного вызова.

1. Укажем в декораторе функции нужные нам элементы ввода и вывода:

```
@app.callback(Output('clustered_map_chart', 'figure'),
              Input('year_cluster_slider', 'value'),
              Input('ncluster_cluster_slider', 'value'),
              Input('cluster_indicator_dropdown', 'value'))
```

2. Объявим функцию с тремя входными параметрами:

```
def clustered_map(year, n_clusters, indicators):
```

3. Создадим экземпляры объектов `SimpleImputer`, `StandardScaler` и `KMeans` для дальнейшей работы с ними. Заметьте, что при создании объекта `SimpleImputer` мы указали, как закодированы пропущенные значения. В нашем случае это `np.nan`, но могут быть и `N/A`, и `0`, и `-1`, и другие значения. Код создания объектов приведен ниже:

```
imp = SimpleImputer(missing_values=np.nan, strategy='mean')
scaler = StandardScaler()
kmeans = KMeans(n_clusters=n_clusters)
```

4. Создадим поднабор данных на основе датафрейма `poverty`, в котором оставим только страны и ограничим данные выбранным годом. В качестве столбцов укажем выбранные индикаторы, а также год и название страны. Код создания датафрейма показан ниже:

```
df = poverty[poverty['is_country'] & poverty['year'].
eq(year)][indicators + ['Country Name', 'year']]
```

5. Объявим отдельный датафрейм только с индикаторами. Причина того, почему нам потребовалось создать два датафрейма, состоит в том, что датафрейм `df` мы будем использовать для отображения данных на карте, и нам понадобятся годы и страны. В то же время в наборе данных `data` будут содержаться только числовые данные, с которыми будет взаимодействовать наша модель:

```
data = df[indicators]
```

6. Бывает, как вы не раз видели в этой книге, что в данных присутствуют полностью пустые колонки. В этом случае мы не можем воспользоваться подстановкой значений, поскольку у нас нет ни средних значений по столбцу, ничего, и что с ним делать – неизвестно. Мне кажется, в таких ситуациях лучше вовсе не строить график и выводить пользователю предупреждение о том, что при выбранном сочетании факторов у нас нет достаточного количества данных для построения модели. Но для начала надо проверить, наш ли это случай. У датафреймов есть метод `isna`, позволяющий преобразовать данные в `True` и `False` в зависимости от наличия пропущенных значений. Чтобы распознать ситуацию, когда все значения в датафрейме пропущены, воспользуемся методом `all`. В результате получим информацию о пропущенных значениях в разрезе столбцов в виде объекта `Series` со значениями `True` и `False`. Проверим, есть ли среди них значения `True`, воспользовавшись методом `any`. Если да, то создаем пустой график с информативным заголовком, говорящим о том, что для выбранной комбинации года и индикаторов нет полной информации:

```
if df.isna().all().any():
    return px.scatter(title='No available data for the selected
combination of year/indicators.')
```

7. Если же все в порядке и у нас нет полностью пустых колонок в наборе, создадим новый датафрейм с заполненными пропусками по среднему, как показано ниже:

```
data_no_na = imp.fit_transform(data)
```

8. Затем нормализуем наши данные из датафрейма `data_no_na`, воспользовавшись объектом `StandardScaler`:

```
scaled_data = scaler.fit_transform(data_no_na)
```

9. Теперь обучим нашу модель на нормализованных данных:

```
kmeans.fit(scaled_data)
```

10. На этом этапе у нас есть все необходимые данные для создания диаграммы, главным образом это атрибут `labels_`. Воспользуемся методом `px.choropleth` для построения картограммы. Здесь для вас не будет ничего нового:

```
fig = px.choropleth(df,
                    locations='Country Name',
                    locationmode='country names',
                    color=[str(x) for x in  kmeans.labels_],
                    labels={'color': 'Cluster'},
                    hover_data=indicators,
                    height=650,
                    title=f'Country clusters - {year}.
Number of clusters: {n_clusters}<br>Inertia: {kmeans.inertia_:,.2f}',
```

После этого добавим географические атрибуты, которые уже использовали ранее, для изменения внешнего вида карты, и адаптируем визуализацию под наше приложение. Полный код функции обратного вызова показан ниже:

```
@app.callback(Output('clustered_map_chart', 'figure'),
              Input('year_cluster_slider', 'value'),
              Input('ncluster_cluster_slider', 'value'),
              Input('cluster_indicator_dropdown', 'value'))
def clustered_map(year, n_clusters, indicators):
    if not indicators:
        raise PreventUpdate
    imp = SimpleImputer(missing_values=np.nan, strategy='mean')
    scaler = StandardScaler()
    kmeans = KMeans(n_clusters=n_clusters)

    df = poverty[poverty['is_country'] & poverty['year'].eq(year)]
[indicators + ['Country Name', 'year']]
    data = df[indicators]
    return
    if df.isna().all().any():
        return px.scatter(title='No available data for the selected
combination of year/indicators.')
    data_no_na = imp.fit_transform(data)
    scaled_data = scaler.fit_transform(data_no_na)
    kmeans.fit(scaled_data)
```

```
    fig = px.choropleth(df,
                        locations='Country Name',
                        locationmode='country names',
                        color=[str(x) for x in  kmeans.labels_],
                        labels={'color': 'Cluster'},
                        hover_data=indicators,
                        height=650,
                        title=f'Country clusters - {year}.
Number of clusters: {n_clusters}<br>Inertia: {kmeans.inertia_:,.2f}',
                        color_discrete_sequence=px.colors.qualitative.T10)
    fig.add_annotation(x=-0.1, y=-0.15,
                       xref='paper', yref='paper',
                       text='Indicators:<br>' + "<br>".join(indicators),
                       showarrow=False)
    fig.layout.geo.showframe = False
    fig.layout.geo.showcountries = True
    fig.layout.geo.projection.type = 'natural earth'
    fig.layout.geo.lataxis.range = [-53, 76]
    fig.layout.geo.lonaxis.range = [-137, 168]
    fig.layout.geo.landcolor = 'white'
    fig.layout.geo.bgcolor = '#E5ECF6'
    fig.layout.paper_bgcolor = '#E5ECF6'
    fig.layout.geo.countrycolor = 'gray'
    fig.layout.geo.coastlinecolor = 'gray'
    return fig
```

В этой главе мы сделали большой шаг в области визуализации и интерактивности данных. Попутно мы рассмотрели один из простейших методов машинного обучения для кластеризации данных. Опции для выбора, которые будут предоставлены пользователю, зависят от конкретной бизнес-задачи. Вы можете сами быть экспертом в этой области и задавать требования, а можете работать совместно со специалистом. Для полноценного анализа данных важна не только визуализация и статистика, но и знание предметной области, а в отношении машинного обучения этот аспект выходит на первый план.

Вы можете самостоятельно продолжить изучение методов машинного обучения. Умение создавать интерактивные дашборды будет отличным подспорьем в освоении различных моделей и позволит вам исследовать тенденции и оперативно принимать решения на основе данных. В результате вы научитесь создавать автоматизированные решения, способные выдавать перечень рекомендаций или самостоятельно принимать решения на основе имеющихся данных.

Теперь давайте вспомним, что конкретно мы узнали из этой главы.

Заключение

Сначала мы познакомились с понятием кластеризации данных. Построили простейшую модель на основе крошечного набора данных. Запустив модель несколько раз, мы смогли оценить ее эффективность при разном количестве кластеров.

Далее мы узнали, что из себя представляет метод локтя, помогающий определить оптимальное количество кластеров для модели. Он состоит в определении точки, в которой добавление кластеров перестает приносить ощутимую прибавку в отношении качества модели. Воспользовавшись этой техникой, мы выполнили кластеризацию стран по выбранным показателям, посмотрев, как изученные методы работают на практике.

После этого познакомились с двумя техниками подготовки данных перед анализом: подстановкой пропущенных значений и масштабированием. Это позволило нам привести исходные данные к виду, пригодному для анализа, и создать на их основе интерактивное приложение, что мы и сделали в конце главы.

В следующей главе мы узнаем, какие дополнительные возможности предоставляют функции обратного вызова. Функции, которые мы писали до этого, были достаточно простыми и стационарными. Но иногда нам требуется создать для пользователей более динамический интерфейс. К примеру, на основании выбранного значения в выпадающем списке может быть выбран тип графика или создаваться другой список. Об этом мы и поговорим в следующей главе.

Глава 10

Ускорение работы приложений с помощью улучшений функций обратного вызова

В этой главе мы постараемся вывести наше приложение на новый уровень абстракции, воспользовавшись дополнительными возможностями, предоставляемыми функциями обратного вызова. До данного момента мы просто предлагали пользователю интерактивные компоненты, с которыми он мог взаимодействовать. Основываясь на наборе предоставляемых компонентами опций, пользователь может осуществлять определенные действия, например строить графики. Здесь же мы рассмотрим другие способы взаимодействия пользователя с приложением, в частности отложенное выполнение функций обратного вызова по нажатии на специальную кнопку. Также мы узнаем, как можно дать пользователю возможность динамически менять макет приложения путем добавления на него компонентов. Мы используем некоторые из этих навыков для внесения важных изменений в наше приложение с кластеризацией, созданное в главе 9.

Сначала мы познакомимся с необязательным элементом функций обратного вызова **State**. В приложениях, которые мы писали до сих пор, любые взаимодействия пользователя с интерактивными элементами ввода немедленно приводили к запуску соответствующей функции обратного вызова. Но зачастую нам нужно, чтобы пользователь сделал свой выбор, а функция запускалась только по нажатии на определенную кнопку. Это бывает важно, когда у вас есть множество элементов управления и вы не хотите, чтобы функция отрабатывала после каждого изменения любого из этих элементов. К тому же выполнение функции может оказаться достаточно длительным, и вам нужно, чтобы она запускалась лишь один раз по окончании редактирования элементов на форме.

Освоив концепцию элемента `State`, мы познакомимся с новым типом динамических обратных вызовов, позволяющим пользователю самому вносить изменения в приложение, к примеру путем добавления нового графика. До этого

момента пользователю было разрешено взаимодействовать только с доступными ему элементами ввода. Динамические компоненты, создаваемые на основе интерактивных действий пользователя, позволяют вывести эту концепцию на новый уровень.

Также мы узнаем, что из себя представляют шаблонные обратные вызовы, позволяющие гибко связывать динамически созданные и интерактивные компоненты.

Темы, которые будут рассмотрены в главе:

- знакомство с элементом `State`;
- создание взаимосвязанных компонентов;
- добавление пользователем динамических компонентов в приложение;
- введение в шаблонные обратные вызовы.

Технические требования

В этой главе мы продолжим использовать пакеты, с которыми познакомились ранее, а главным образом сосредоточимся на дополнительных возможностях функций обратного вызова. Мы будем применять в работе пакеты Dash, Dash Core Component, Dash HTML Components и Dash Bootstrap Components. Для манипулирования данными воспользуемся пакетом pandas. Библиотеки Plotly и Plotly Express помогут нам при визуализации данных, а пакет JupyterLab позволит независимо опробовать новый функционал перед внедрением его в приложение.

Исходный код к этой главе располагается в хранилище GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_10**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3v6ZYJw**.

Знакомство с элементом State

Типичная структура функций обратного вызова, с которыми мы работали до этого момента, включала в себя один или несколько элементов ввода и/или вывода (`Input` и `Output` соответственно). При этом функция автоматически запускалась всякий раз, когда пользователь изменял значение элемента ввода. Теперь мы бы хотели немного ослабить это требование. Начнем с простого примера, демонстрирующего, зачем и как следует использовать элемент `State`, являющийся необязательным параметром функции обратного вызова.

Взгляните на рис. 10.1, чтобы понять, какая задача перед нами стоит.

Как видите, в строке вывода под текстовым полем отображается неправильная информация. Причина в том, что мы искусственно заложили паузу в работу приложения, чтобы смоделировать работу реального приложения. На самом деле вывод не то что неправильный, он просто немного запаздывает. И в таких приложениях, как это, где вывод зависит от нескольких элементов ввода, это бывает очень важно. Задержка, с которой мы вводим значения в поля, может приводить к таким неожиданным результатам.

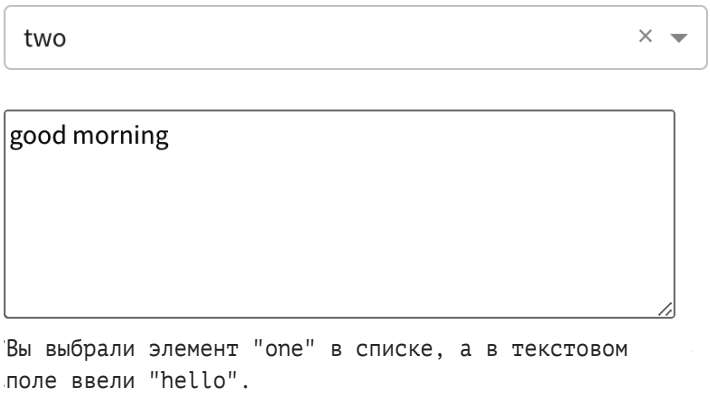


Рис. 10.1. Интерактивное приложение с запаздывающей синхронизацией элементов

Еще более серьезной проблемой является то, что эти выборы могут отнимать достаточно много времени и ресурсов. У нас очень маленький набор данных, над которым мы производим довольно примитивные действия, что не дает нам повода для волнения. На практике вы будете сталкиваться с гораздо более объемными датасетами, и вычисления будут куда более сложными. К примеру, изменить количество кластеров в нашей модели можно мгновенно. В реальных задачах на это может потребоваться несколько секунд или минут. Мы решим эту задачу при помощи кнопки подтверждения **Submit** и нового для наших функций обратного вызова элемента `State`.

Элементы управления в виде кнопки доступны в пакете Dash HTML Components. Также они есть в пакете Dash Bootstrap Components. Использование последнего дает два преимущества. Во-первых, кнопки из пакета Dash Bootstrap Components хорошо визуально интегрируются в тему приложения. Во-вторых, что более важно, их легко можно окрасить в ключевые системные цвета для передачи некоторых сообщений, таких как «success» (успех), «warning» (предупреждение) и «danger» (опасность).

Для добавления кнопок в интерфейс можно воспользоваться классами `dcc.Button` или `dbc.Button`.

Но для начала посмотрим, чем отличаются элементы `Input` и `State`.

Различия между элементами Input и State

Первое, что необходимо понимать, – это то, что изменение элемента ввода `Input` запускает функцию, а в элементах `State` просто хранится некоторое состояние приложения. Мы сами вправе решать, какие компоненты будут маркироваться как элементы `State`, а какие – как `Input`.

Давайте обобщим инструкции для создания соответствующих декораторов функций обратного вызова:

- элементы в декораторе функции должны следовать в порядке `Output`, `Input`, а затем необязательные `State`. Если одному элементу соответствуют несколько компонентов, они должны следовать подряд;

- элемент State является необязательным;
- запуск функции осуществляется по изменению элемента Input. Изменение компонентов, помеченных как State, не приводит к немедленному выполнению функции;
- при изменении элемента Input функция будет вызвана с актуальными на данный момент значениями элементов State.

Давайте посмотрим на код приложения, вывод которого показан на рис. 10.1, после чего исправим его так, чтобы он соответствовал нашим ожиданиям. На данный момент текст функции выглядит так:

```
@app.callback(Output('output', 'children'),
              Input('dropdown', 'value'),
              Input('textarea', 'value'))
def display_values(dropdown_val, textarea_val):
    time.sleep(4)
    return f'Вы выбрали элемент "{dropdown_val}" в списке,
а в текстовом поле ввели "{textarea_val}".'
```

Обратите внимание, что функция будет запускаться после изменения любого элемента ввода. Изменение, которое мы хотим внести, потребует от нас двух шагов.

1. Добавим кнопку на страницу и разместим ее под текстовым полем:

```
import dash_bootstrap_components as dbc
dbc.Button("Submit", id="button")

# ИЛИ

import dash_html_components as html
html.Button("Submit", id="button")
```

2. Добавим элемент Input в нашу функцию. Попутно введем новое свойство n_clicks, с которым раньше не сталкивались. Как ясно из названия, в этом свойстве указано количество щелчков, которое было сделано на соответствующем компоненте за время его жизни в рамках пользовательской сессии. С каждым щелчком этот показатель увеличивается на единицу, и эту переменную можно использовать для проверки и контроля поведения обратного вызова. Заметьте, что мы также можем дать этому показателю значение по умолчанию – обычно оно равно нулю, но может быть и произвольной величиной:

```
Input("button", "n_clicks")
```

3. Теперь, когда мы назначили кнопку в качестве элемента Input, нам нужно сохранить компоненты Dropdown и Textarea, но сопоставить их с элементами State, как показано ниже:

```
@app.callback(Output('output', 'children'),
              Input('button', 'n_clicks'),
              State('dropdown', 'value'),
              State('textarea', 'value'))
```

4. В результате произведенных изменений функция обратного вызова будет запускаться только при изменении количества щелчков по кнопке, поскольку только для нее задан параметр `Input`. Пользователь может сколько угодно раз выбирать элементы из выпадающего списка и заполнять текстовое поле – пока не будет нажата кнопка, функция обратного вызова запущена не будет.

5. При первой загрузке приложения свойство `n_clicks` приобретает значение `None`, заданное по умолчанию. При этом в текстовом поле нет заполнения, а в выпадающем списке ничего не выбрано. Что ж, сделаем как обычно – используем инструкцию `raise PreventUpdate`, если в параметре `n_clicks` нет значения. Теперь внесем изменения в объявление функции. Заметьте, что мы добавили параметр `n_clicks`, поставив его первым:

```
def display_values(n_clicks, dropdown_val, textarea_val):
    if not n_clicks:
        raise PreventUpdate
    return ...
```

Теперь приложение будет выглядеть так, как показано на рис. 10.2. И управляться с ним будет легко и удобно.

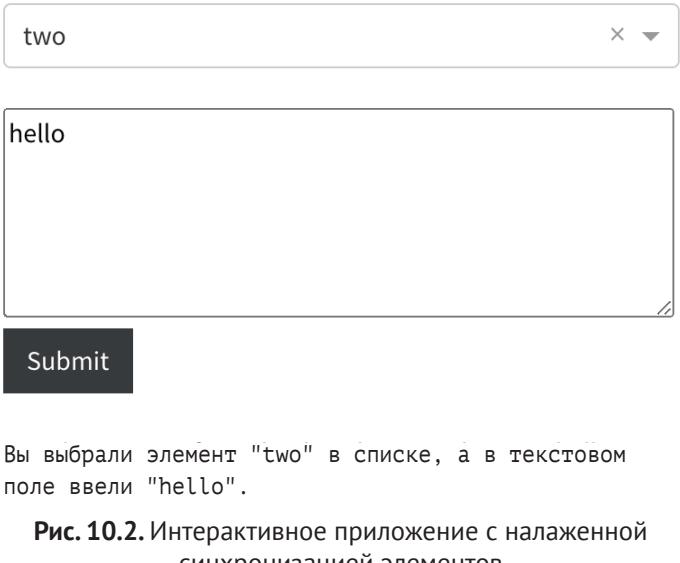


Рис. 10.2. Интерактивное приложение с налаженной синхронизацией элементов

Можно дополнить приложение визуальным эффектом, говорящим о том, что выполняется какой-то процесс.

Давайте продемонстрируем это на примере нашего приложения с кластеризацией данных из предыдущей главы. На рис. 10.3 показан эффект, которого мы хотим добиться.

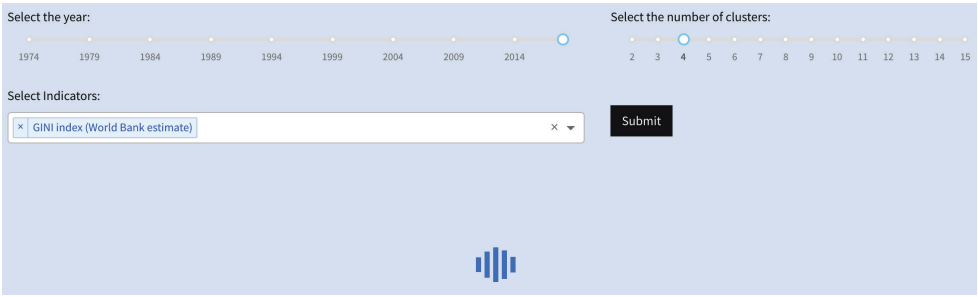


Рис. 10.3. Кластеризация с кнопкой **Submit** и индикатором прогресса

Как видите, здесь мы внедрили сразу две новинки. Во-первых, появилась кнопка **Submit**, о которой мы только что говорили. Во-вторых, мы использовали компонент `Loading`, входящий в состав библиотеки Dash Core Components. Он представляет собой анимированную иконку, говорящую пользователю о том, что идет формирование отчета. Использовать этот компонент очень просто, а пользователю будет полезно знать, что ничего не зависло, а данные уже спешат к нему. Если честно, я бы применял компонент `Loading` для всех элементов вывода, чтобы использовать приложение было комфортно. Ниже показан фрагмент кода с этим компонентом:

```
import dash_core_components as dcc
dcc.Loading([
    dcc.Graph(id='clustered_map_chart')
])
```

Мы просто перенесли существующий элемент `Graph` внутрь компонента `Loading`, указав его в качестве аргумента `children`. Это приведет к появлению анимированного индикатора, который исчезнет только после вывода на экран графика.

Давайте внесем необходимые изменения в объявление функции:

```
@app.callback(Output('clustered_map_chart', 'figure'),
              Input('clustering_submit_button', 'n_clicks'),
              State('year_cluster_slider', 'value'),
              State('ncluster_cluster_slider', 'value'),
              State('cluster_indicator_dropdown', 'value'))
def clustered_map(n_clicks, year, n_clusters, indicators):
```

Мы сделали два изменения. Добавили кнопку с идентификатором `clustering_submit_button` в качестве элемента ввода `Input`, а для остальных компонентов поменяли ключевое слово `Input` на `State`. Также мы передали `n_clicks` на вход функции первым аргументом. Помните, что называться аргументы

могут произвольно, важен лишь порядок их следования. Мы дали аргументам говорящие имена, так что сможем спокойно обращаться к ним в теле функции.

Итак, мы изменили функционал кластеризации в приложении, дав пользователю больше свободы и улучшив удобство работы за счет компонента `Loading`. Советуем использовать этот компонент везде, где это возможно.

Ну а мы двигаемся дальше, на новый уровень абстракции!

Создание взаимосвязанных компонентов

Как насчет того, чтобы добавить в приложение интерактивный компонент, значения которого, заданные пользователем, будут служить входными элементами функции, которая, в свою очередь, будет отвечать за их вывод в другом компоненте? Эта замысловатая структура показана на рис. 10.4, а далее мы опишем ее более подробно.

Рис. 10.4. Приложение, в котором один компонент динамически определяет значения другого

Давайте пройдемся по визуальным элементам этого приложения.

1. **Сообщение об успехе.** Зеленая полоса с текстом в верхней части страницы не появляется при загрузке приложения. Она показывается только тогда, когда пользователь вводит опции в списке и нажимает на кнопку **Set options**. Заметьте, что текст этого сообщения генерируется динамически – в нем перечислены опции, которые ввел пользователь. Также обратите внимание, что это сообщение можно закрыть крестом, расположенным в его правой части.

2. **Строка с инструкцией.** Обычное текстовое сообщение, указывающее на возможность ввести текст в компонент `Textarea`, который будет в дальнейшем использоваться в качестве свойства `options` компонента `Dropdown`, располагающегося ниже. В данный момент выпадающий список пуст, и на графике, располагающемся под ним, также не выводится никакая информация.
3. **Кнопка Set options.** После ввода опций в текстовое поле и нажатия на эту кнопку они станут пунктами для выбора в выпадающем списке.
4. **Результирующий график.** В данном случае мы использовали трехбуквенные коды стран. Пользователь вводит сокращения для стран, которые в дальнейшем сможет выбрать из списка. После этого выполняется фильтрация исходного набора данных по выбранной стране и выводится график с информацией о ней.

В подобном приложении нет никакого практического смысла, поскольку гораздо проще было бы всего лишь заполнить выпадающий список всеми странами. Мы просто хотим продемонстрировать вам продвинутые возможности на уже знакомом вам наборе данных. К тому же здесь есть определенные предпосылки для возникновения ошибок. Что, если пользователь не знает трехбуквенного кода для нужной ему страны? Или допустит опечатку? Опять же, это просто демонстрационный пример.

Скрипт этого приложения уместится всего в 30 строк, но при этом в нем будет целых два зависимых слоя, один из которых будет заполняться на основании данных из другого.

Давайте создадим макет будущего приложения, после чего напишем две функции обратного вызова, которые позволят добавить ему интерактивности.

1. Импортируем необходимые пакеты:

```
from jupyter_dash import JupyterDash
import dash_core_components as dcc
import dash_html_components as html
import dash_bootstrap_components as dbc
import pandas as pd
poverty = pd.read_csv('data/poverty.csv')
```

2. Создадим макет приложения. Все последующие компоненты будут частью этого макета:

```
app = JupyterDash(__name__, external_stylesheets=[dbc.themes.COSMO])

app.layout = html.Div([
    component_1,
    component_2
...
])
```

3. Создадим пустой элемент `div` для сообщения об успехе:

```
html.Div(id='feedback')
```

4. Добавим компонент `Label` с указанием о том, как обращаться с приложением:

```
dbc.Label("Create your own dropdown, add options one per line:")
```

5. Добавим компонент `Textarea`. Обратите внимание, что в пакете Dash Core Components есть точно такой же компонент:

```
dbc.Textarea(id='text', cols=40, rows=5)
```

6. Создадим кнопку для наполнения списка элементами:

```
dbc.Button("Set options", id='button')
```

7. Теперь пришло время выпадающего списка:

```
dcc.Dropdown(id='dropdown')
```

8. Завершим наш макет компонентом `Graph`:

```
dcc.Graph(id=chart')
```

Этого достаточно для визуального наполнения нашего приложения. Осталось дополнить его двумя функциями обратного вызова для обеспечения интерактивности.

1. `set_dropdown_options`: функция будет принимать на вход содержимое поля `Textarea` и возвращать список элементов для добавления в выпадающий список.
2. `create_population_chart`: функция будет принимать содержимое выпадающего списка и генерировать график с численностью населения по выбранной стране.

Начнем с первой функции.

1. Напишем декоратор с нужными элементами `Output`, `Input` и `State`. Эта наша функция будет влиять на два компонента: выпадающий список и сообщение об успехе. В качестве элемента `Input` будет выступать кнопка, а в качестве элемента `State` – компонент `Textarea`:

```
@app.callback(Output('dropdown', 'options'),
              Output('feedback', 'children'),
              Input('button', 'n_clicks'),
              State('text', 'value'))
```

2. Создадим объявление функции с подходящими именами аргументов:

```
def set_dropdown_options(n_clicks, options):
```

3. Объявим переменную, в которой будем хранить введенный текст как список. Это можно сделать, применив метод `split` к содержимому текстового поля. Также не забудем выполнить проверку на отсутствие щелчков по кнопке:

```
if not n_clicks:
    raise PreventUpdate
text = options.split()
```

4. Создадим сообщение об успехе в виде объекта `Alert`, доступного в пакете Dash Bootstrap Components. Обратите внимание, что в качестве параметра `color` мы передаем значение `"success"`. Здесь вы могли бы расширить логику и проверять введенный текст на корректность, используя при необходимости для параметра `color` значения `"warning"` или `"danger"`. Заметьте также, что мы динамически добавляем в сообщение список введенных стран через запятую при помощи метода `join` и передаем параметр `dismissable=True`, чтобы пользователь при желании мог закрыть сообщение крестиком.

   Соберем список стран для параметра `options` компонента `Dropdown`. Для этого пройдемся по списочной переменной `text` и создадим соответствующие словари:

```
options = [{'label': t, 'value': t} for t in text]
```

5. Вернем из функции кортеж из `options` и `message`:

```
return options, message
```

Полный код первой функции:

```
@app.callback(Output('dropdown', 'options'),
              Output('feedback', 'children'),
              Input('button', 'n_clicks'),
              State('text', 'value'))
def set_dropdown_options(n_clicks, options):
    if not n_clicks:
        raise PreventUpdate
    text = options.split()
    message = dbc.Alert(f"Success! you added the options: {', '.join(text)}",
                        color='success',
                        dismissable=True)
    options = [{'label': t, 'value': t} for t in text]
    return options, message
```

Теперь приступим к написанию второй функции, которая должна принимать на вход код страны и использовать его для построения диаграммы.

1. Напишем декоратор для функции с одним элементом `Output` и одним `Input`:

```
@app.callback(Output('chart', 'figure'),
              Input('dropdown', 'value'))
```

2. Объявим функцию и проверим переданное значение:

```
def create_population_chart(country_code):
    if not country_code:
        raise PreventUpdate
```

3. Ограничим датафрейм на основе входящего аргумента:

```
df = poverty[poverty['Country Code']==country_code]
```

4. Возвратим готовый график:

```
return px.line(df,
               x='year',
               y='Population, total',
               title=f"Population of {country_code}")
```

Теперь можно проверять приложение на интерактивность.

Чтобы лучше разобраться в том, как работает наше приложение, и отследить его внутренние процессы, можно запустить его в режиме отладки, выполнив инструкцию `app.run_server(debug=True)`, и посмотреть, как взаимодействуют компоненты. Соответствующий граф показан на рис. 10.5.

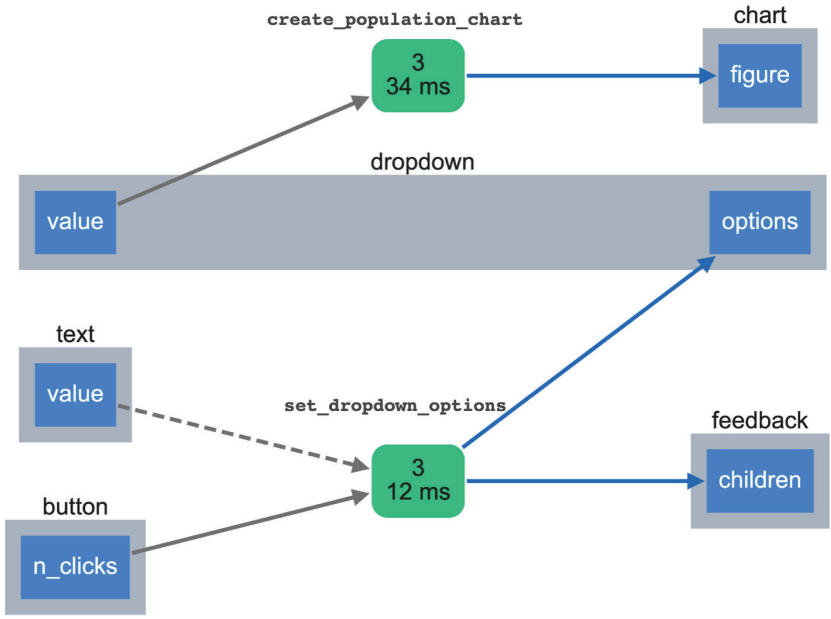


Рис. 10.5. Граф обратных вызовов приложения

Вы видите на графе имена компонентов и их идентификаторы, указанные в коде, и можете легко отследить последовательность событий, начиная с левого нижнего угла по стрелкам в сторону правого верхнего.

Итак, мы рассмотрели на примере, как можно создавать динамические зависимости между компонентами в приложении. Dash прекрасно справляется с подобными задачами, и все функции отрабатывают в точности тогда и так, как нами задумано.

Теперь сделаем еще один шаг в мир абстракции и позволим пользователю самому добавлять полноценные компоненты в приложение по нажатии на кнопку!

Добавление пользователем динамических компонентов в приложение

Пользователи могут не только добавлять компоненты в приложение в процессе его работы, но и динамически генерировать их содержимое. Взгляните на рис. 10.6, на котором изображена простейшая схема добавления в приложение компонентов по нажатии на кнопку.

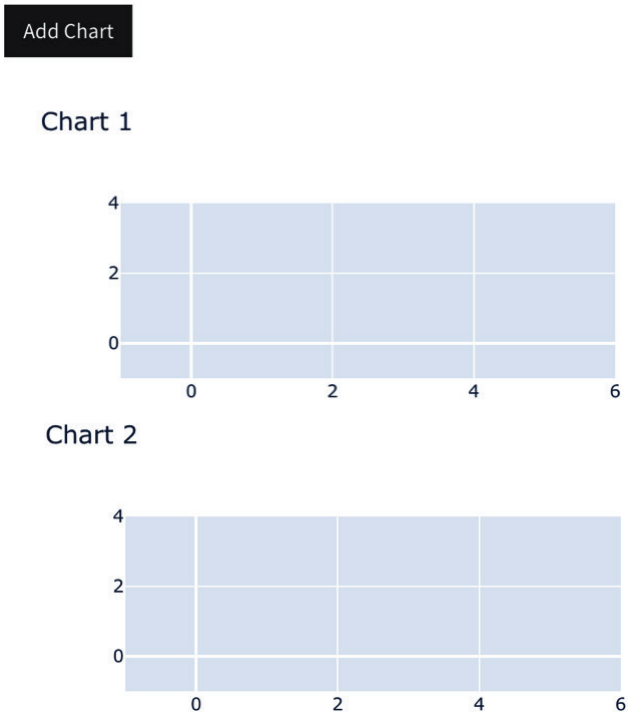


Рис. 10.6. Компоненты, добавленные в приложение пользователем

Несмотря на простоту приложения, динамически добавленные графики имеют собственные уникальные заголовки, построенные на основании свойства `n_clicks`, значение которого монотонно возрастает с каждым щелчком.

Для создания такого приложения понадобится написать совсем немного строк кода. Давайте начнем с создания макета, который будет содержать два компонента.

1. Создадим кнопку для добавления нового графика:

```
dbc.Button("Add Chart", id='button')
```

2. Теперь добавим элемент `div` с атрибутом `children`, в котором будет находиться пустой список. Именно с этим списком мы и будем впоследствии работать:

```
html.Div(id='output', children=[])
```

Во время первого запуска приложения пользователь видит только одну кнопку, предназначенную для добавления новых графиков. При каждом нажатии на кнопку в нижней части будет добавляться пустой график.

Что ж, давайте напишем функцию обратного вызова, чтобы оживить наше приложение.

1. Сначала, как и всегда, напишем декоратор с правильно расставленными элементами `Output`, `Input` и `State`. Любопытно отметить, что в данном случае свойство `children` одного и того же компонента `div` будет задействовано и в качестве элемента `Output`, и в качестве `State`. Обычно мы принимали на вход функции значение одного компонента и както влияли на содержимое другого компонента. А кто сказал, что мы не можем взять компонент, произвести в нем изменения и вернуть его в новом состоянии? Именно это мы и сделаем:

```
@app.callback(Output('output', 'children'),
              Input('button', 'n_clicks'),
              State('output', 'children'))
```

2. Объявим функцию и проверим значение `n_clicks`. Обратите внимание, что атрибут `children` здесь выступает как элемент `State`:

```
def add_new_chart(n_clicks, children):
    if not n_clicks:
        raise PreventUpdate
```

3. Создадим пустую столбчатую диаграмму с динамическим заголовком, основанным на свойстве `n_clicks`:

```
new_chart = dcc.Graph(figure=px.bar(title=f"Chart {n_clicks}"))
```

4. Добавим созданный график в списочную переменную `children`. Если помните, в элементе `div` мы разместили атрибут `children` с пустым списком. В показанной ниже строке кода мы добавляем к этому списку `new_chart`. Здесь нет ничего нового, мы просто воспользуемся методом `append`:

```
children.append(new_chart)
```

5. Теперь, когда мы изменили список `children`, мы можем вернуть его обратно. Помните, что возвращаемое из функции значение отправится в тот же элемент `div`, который на этот раз выступает как элемент вывода:

```
return children
```

Обратите внимание, что при создании этого приложения мы пользовались уже знакомыми вам принципами и концепциями, не введя ни одного нового понятия. С одной стороны, мы передали атрибут `children` в функцию обратного вызова и вернули его же, а с другой – использовали свойство `n_clicks` для динамического заполнения заголовков графиков.

На рис. 10.7 показаны взаимосвязи между созданными элементами.

Рис. 10.7. Граф взаимодействий элементов внутри функции

Эта диаграмма не изменится, сколько бы графиков мы ни добавили в приложение. Так что вам не нужно писать отдельные функции для каждого нажатия на кнопку.

Если хотите, мы можем пойти еще дальше и добавить под каждый график свой выпадающий список для настройки содержимого диаграммы. При этом каждый список будет работать независимо от остальных и влиять на содержимое только одного графика, под которым находится. Хорошая новость состоит в том, что для всего этого нам понадобится всего одна дополнительная функция, использующая шаблоны обратного вызова.

Введение в шаблонные обратные вызовы

Знакомство с этой концепцией – а в этом разделе мы действительно познакомимся с новым для вас подходом – позволит вывести ваши приложения на совершенно новый уровень интерактивности. Польза описанной здесь техники кроется в возможности управлять интерактивностью компонентов, которые даже не были созданы. В предыдущем разделе мы позволили пользователю создавать собственные компоненты, которых до этого в приложении не существовало. Любопытно, что функцию обратного вызова, обрабатывающую все эти новые компоненты, написать будет довольно просто – от созданных ранее она будет отличаться лишь тем, что идентификаторы обрабатываемых компонентов будут не статическими, а динамическими.

До этого момента мы задавали атрибут `id` для компонентов в виде строк, и единственным требованием, применяющимся к ним, была их уникальность. В этом разделе мы познакомимся с новым подходом к указанию идентификаторов – с использованием словарей. Давайте сначала посмотрим на результат, к которому будем стремиться, после этого изменим макет, функции обратного вызова и поговорим о самой концепции указания идентификаторов элемен-

тов посредством словарей. На рис. 10.8 показано, как должно выглядеть наше приложение.

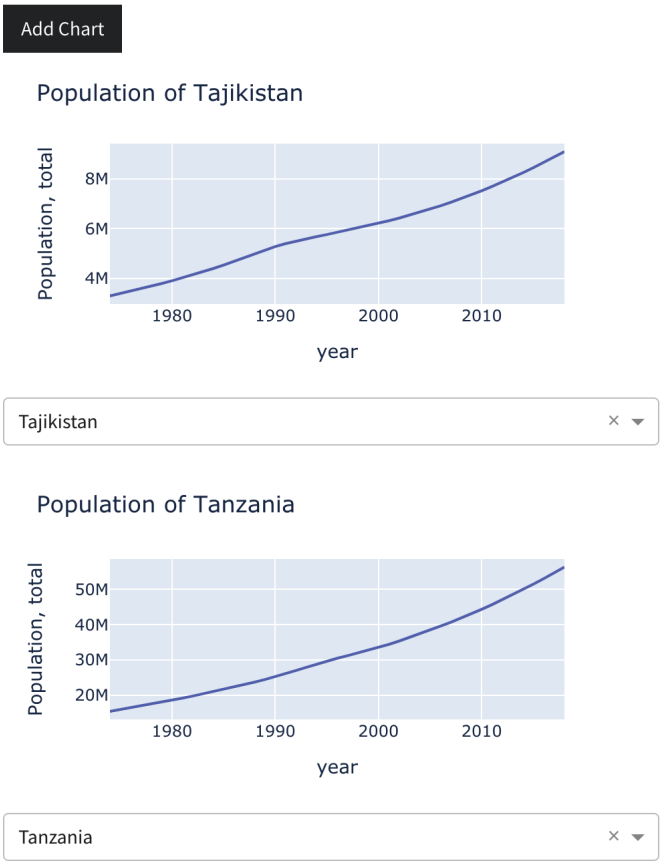


Рис. 10.8. Приложение с возможностью добавления интерактивных компонентов

В нашем предыдущем приложении пользователь мог генерировать новые компоненты на лету и даже динамически задавать их содержимое, как в случае с заголовком на основе атрибута `n_clicks`. Однако после добавления компонентов в приложение пользователь не имел возможности с ними взаимодействовать. Иными словами, эти компоненты могли создаваться динамически, включая их содержимое, но после создания становились полностью статическими, без возможности взаимодействия с ними.

В этом разделе мы сделаем динамически создаваемые графики полностью интерактивными и свяжем их с конкретными выпадающими списками. Как видно на рис. 10.8, каждый график идет в комплекте со списком, и пользователь может создавать их независимо друг от друга и сравнивать результаты. Выбирая разные страны, пользователь будет видеть графики по ним. Это простейший пример, но вы можете себе представить и гораздо более сложные реализации с использованием этого подхода.

Представленный здесь новый функционал можно внедрить за три простых шага.

1. Изменим функцию `add_new_chart` таким образом, чтобы под графиком вставлялся выпадающий список и добавлялся не один компонент, как раньше, а два. Обратите внимание, что макет приложения не изменился, у нас по-прежнему есть одна кнопка и пустой элемент `div`.
2. Напишем новую функцию обратного вызова, с помощью которой свяжем выпадающий список с графиком внутри каждого добавляемого блока.
3. Изменим подход к указанию идентификаторов компонентов в приложении. В этом кроется вся магия нового подхода, позволяющего управлять новыми компонентами и их взаимодействиями с другими компонентами в рамках единственной функции.

Начнем с изменения функции `add_new_chart`.

1. Ранее в функции мы объявили переменную `new_chart`, и она останется и в новой реализации. Под ней добавим еще одну переменную `new_dropdown` – для выпадающего списка, из которого пользователь сможет выбрать страну для визуализации на графике:

```
new_chart = dcc.Graph(figure=px.bar(title=f"Chart {n_clicks}"))
countries = poverty[poverty['is_country']]['Country Name'].
drop_duplicates().sort_values()
new_dropdown = dcc.Dropdown(options=[{'label': c, 'value': c}
                                     for c in countries])
```

2. Добавим новый компонент. В первом примере мы создавали только `new_chart`, на этот раз добавим два созданных компонента. Для удобства мы создадим элемент `div` и разместим компоненты в нем, а уже сам `div` добавим в список. Таким образом, мы добавим один контейнер, содержащий два элемента:

```
children.append(html.Div([
    new_chart,
    new_dropdown
]))
```

Этого достаточно, чтобы по нажатии на кнопку в приложении появлялись график и выпадающий список под ним. Как видите, ничего особенного мы не сделали. Вскоре мы вернемся сюда и изменим подход к указанию идентификаторов созданных компонентов.

Итак, по нажатии на кнопку в приложение добавляются целые блоки, состоящие из графика и выпадающего списка. Очевидно, что при определении связи между ними один из компонентов (график) должен стать элементом `Output`, а другой (список) – элементом `Input`. Таким образом, нам понадобится функция обратного вызова. Давайте создадим ее, а затем посмотрим, как нам

связать динамические идентификаторы компонентов и определить взаимодействие между функциями. Новая функция будет ничем не сложнее любой другой. Ниже приведен ее текст, но без декоратора, который обсудим позже:

```
def create_population_chart(country):
    if not country:
        raise PreventUpdate
    df = poverty[poverty['Country Name']==country]
    fig = px.line(df,
                  x='year', y='Population, total',
                  title=f'Population of {country}')
    return fig
```

На рис. 10.9 показана диаграмма нашей шаблонной функции обратного вызова.

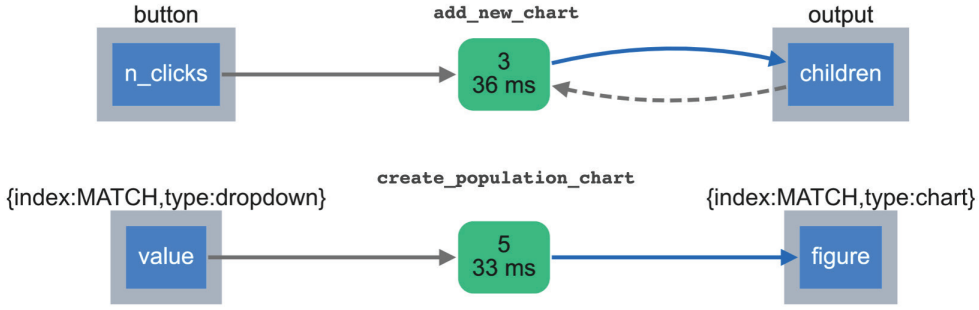


Рис. 10.9. Граф обратного вызова шаблонной функции

На верхнем графике мы видим то же взаимодействие, что и на рис. 10.7, здесь мы просто добавляем компоненты в пустой элемент `div`. Обратите внимание, что над каждым блоком на диаграмме написан идентификатор компонента. На верхнем графе это **button** и **output**.

Для второй функции структура взаимодействий оказалась похожей, но вместо строковых идентификаторов над компонентами присутствуют словари.

Функция обратного вызова использует эти словари для установки соответствий между элементами. Давайте распакуем эти словари и посмотрим, как это работает.

Первый словарь выглядит так: `{"index": MATCH, "type": "dropdown"}`. Уверен, с ключом `type` вам все понятно. Мы используем его для идентификации элементов с таким типом на странице. Важно отметить, что здесь можно использовать произвольные слова для ключа и значения, а не только `type` и `dropdown`, но лучше наделять их каким-то смыслом, чтобы потом было легче ориентироваться. Для второго компонента тип указан `chart`. Опять же, вы можете использовать любое другое слово, но соблюдайте последовательность.

Нам необходимо, чтобы функция отрабатывала независимо для каждой пары компонентов. Иными словами, пользователь должен иметь возможность выбрать в любом из добавленных выпадающих списков нужную ему страну и

посмотреть результаты по ней на одном конкретном графике, не затрагивая остальные. Как этого добиться? Мы просто говорим фреймворку Dash установить соответствие между компонентами при помощи ключевого слова MATCH. И если ключ index, как и type, является произвольным, то слово MATCH зарезервировано и присутствует в модуле dash.dependencies. Есть еще ключевые слова ALL и ALLSMALLER, работающие несколько иначе, но мы сосредоточимся на слове MATCH. Давайте посмотрим, какие изменения нужно внести в функции, чтобы они могли работать с шаблонными идентификаторами. К счастью, нам придется обновить только атрибуты id наших компонентов и передать их в соответствующую функцию.

В функции add_new_chart в качестве атрибутов id внутренних компонентов, которые пользователь добавляет в приложение по нажатии на кнопку, укажем словари. Обратите внимание, что ключу index мы присвоили значение n_clicks. Как мы уже знаем, это динамическая переменная, значение которой меняется при каждом очередном нажатии на кнопку. Это означает, что при добавлении нового блока компонентов в приложение мы каждый раз будем получать уникальный идентификатор, по которому в дальнейшем можно будет найти нужный компонент:

```
def add_new_chart(n_clicks, children):
    new_chart = dcc.Graph(id={'type': 'chart', 'index': n_clicks}, figure=px.bar())
    new_dropdown = dcc.Dropdown(id={'type': 'dropdown', 'index': n_clicks},
                                options=[option_1, option_2,
...])
```

Теперь нужно прописать соответствия идентификаторов во второй функции, отвечающей за интерактивность компонентов. Ключ type мы используем для установки соответствия между компонентами chart и dropdown. Что касается переменной n_clicks, поскольку она динамическая, мы установим соответствие для нее с помощью ключевого слова MATCH, как показано ниже:

```
@app.callback(Output({'type': 'chart', 'index': MATCH}, 'figure'),
              Input({'type': 'dropdown', 'index': MATCH}, 'value'))
def create_population_chart(country):
...
```

Посмотрим на полный код обеих функций:

```
from dash.dependencies import Output, Input, State, MATCH

@app.callback(Output('output', 'children'),
              Input('button', 'n_clicks'),
              State('output', 'children'))
def add_new_chart(n_clicks, children):
    if not n_clicks:
        raise PreventUpdate
```

```
    new_chart = dcc.Graph(id={'type': 'chart', 'index': n_clicks},
                          figure=px.bar(height=300, width=500,
                                        title=f"Chart {n_clicks}"))

    new_dropdown = dcc.Dropdown(id={'type': 'dropdown', 'index': n_clicks},
                                options=[{'label': c, 'value': c}
                                         for c in poverty[poverty['is_country']]
['Country Name'].drop_duplicates().sort_values()])

    children.append(html.Div([
        new_chart, new_dropdown
    ]))
    return children

@app.callback(Output({'type': 'chart', 'index': MATCH}, 'figure'),
              Input({'type': 'dropdown', 'index': MATCH}, 'value'))
def create_population_chart(country):
    if not country:
        raise PreventUpdate
    df = poverty[poverty['Country Name']==country]
    fig = px.line(df,
                  x='year', y='Population, total',
                  title=f'Population of {country}')
    return fig
```

Несложно понять, насколько гибкие и универсальные приложения можно писать с помощью этого функционала, не говоря о легкости, с которой происходит настройка обратных вызовов. В то же время эта техника не самая очевидная, и к ней необходимо привыкнуть. Но игра стоит свеч!

В этой главе мы обсудили многие важные особенности и концепции, применимые к функциям обратного вызова, показали несколько трюков и рассмотрели новый функционал. Давайте подытожим прочитанное.

Заключение

Сначала мы познакомились с элементом декоратора функций обратного вызова `State`. Мы узнали, как в комбинации с элементами `Input` он может осуществлять отсрочку запуска функции до того момента, когда это будет необходимо пользователю. Также мы рассмотрели несколько примеров выполнения функций по нажатии на кнопку. После этого создали простое приложение, в котором пользовательский ввод в одном компоненте динамически влиял на содержимое другого компонента, находившегося в состоянии ожидания. Это содержимое впоследствии может быть использовано для создания другого компонента.

Далее мы рассмотрели технику, позволяющую пользователю добавлять в приложение компоненты с динамическим содержимым.

И наконец, познакомились с наиболее гибким и мощным приемом, позволяющим связывать компоненты с использованием шаблонов. Мы создали приложение, в котором пользователь может добавлять столько блоков с компонентами, сколько хочет. При этом добавленные блоки будут функционировать независимо друг от друга.

В следующей главе мы еще больше расширим функционал наших приложений за счет разделения их на страницы/ссылки. Это позволит разместить в приложении столько компонентов, сколько вы пожелаете.

Глава 11

Ссылки и многостраничные приложения

До сих пор мы размещали все компоненты на единственной странице приложения. Мы просто добавляли графики и интерактивные элементы в созданный элемент `div` один под другим по мере необходимости. Добавление в приложение ссылок (URL) может помочь сэкономить пространство и избежать ситуации переполнения страницы компонентами. Кроме того, с помощью них можно классифицировать содержимое приложения и добавить необходимый контекст, чтобы пользователь понимал, где он находится и что делает.

Еще более интересной является возможность программно добавлять страницы в приложение, просто отображая содержимое на основе ссылки. Именно этим всем мы и займемся в данной главе.

Познакомившись с компонентами **Location** и **Link**, мы немного изменим структуру нашего приложения с помощью создания и изолирования макетов. Вы увидите, как просто можно создавать многостраничные приложения в Dash. Мы разместим основной макет с пустой областью посередине, а содержимое будет отображаться на основе ссылок.

Весь функционал, который мы внедряли в наших приложениях, был основан на индикаторах. Мы создали множество графиков по ним, продемонстрировав их изменчивость во времени и по странам. Но нашим пользователям могут быть интересны и отчеты по странам. В этой связи мы создадим страницы для каждой страны, на которых пользователь сможет анализировать нужные ему индикаторы и при желании сравнивать их с другими странами. Внеся всего несколько изменений, мы добавим целых 169 новых страниц в наше приложение.

Темы, которые будут рассмотрены в главе:

- знакомство с компонентами `Location` и `Link`;
- извлечение и использование атрибутов ссылок;
- разбор ссылок и использование их составляющих для изменения приложения;
- адаптирование приложения под множественные макеты;
- добавление динамически сгенерированных ссылок в приложение;
- внедрение в приложение интерактивности на основе ссылок.

Технические требования

Новые компоненты, с которыми мы познакомимся в этой главе, содержатся в тех же библиотеках, с которыми мы работали ранее: Dash, Dash Core Components, Dash HTML Components и Dash Bootstrap Components. Для манипулирования данными продолжим использовать пакет pandas, библиотеки Plotly и Plotly Express помогут нам при визуализации данных, а пакет JupyterLab мы будем использовать для проверки нового функционала перед его внедрением в приложение.

Главным образом в этой главе мы сосредоточимся на манипулировании ссылками и использовании их в качестве элементов ввода при управлении другими компонентами. Это ничем не будет отличаться от подхода к иным элементам приложений. А начнем со знакомства с двумя важными компонентами, позволяющими реализовать задуманный нами функционал.

Исходный код к этой главе располагается в хранилище GitHub по адресу **https://github.com/PacktPublishing/Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/tree/master/chapter_11**.

Сопроводительные видеофрагменты к этой главе можно посмотреть по адресу **https://bit.ly/3eks3GI**.

Знакомство с компонентами Location и Link

Компоненты `Location` и `Link` входят в состав пакета Dash Core Components, и их назначение напрямую вытекает из их названий. Первый из них – `Location` – ссылается на адресную строку браузера. Обычно компонент `Location` в приложении не обладает собственным визуальным отображением. Главным образом мы обращаемся к нему с просьбой подсказать, где именно в приложении мы находимся, чтобы на основании этой информации выполнить определенные действия. Давайте начнем с простейшего примера, чтобы понять, как это все работает.

1. Создадим простейшее приложение:

```
import dash_html_components as html
import dash_core_components as dcc
from jupyter_dash import JupyterDash
from dash.dependencies import Output, Input
app = JupyterDash(__name__)
```

2. Разместим в приложении макет с компонентом `Location` и пустым элементом `div` под ним:

```
app.layout = html.Div([
    dcc.Location(id='location'),
    html.Div(id='output')
])
```

3. Напишем функцию обратного вызова, принимающую на вход атрибут `href` компонента `Location` и выводящую его в качестве содержимого в элементе `div`:

```
@app.callback(Output('output', 'children'),
              Input('location', 'href'))
def display_href(href):
    return f"You are at: {href}."
```

4. Запустим приложение и увидим следующий вывод:

```
app.run_server(mode='inline')
You are at: http://127.0.0.1:8050/.
```

Все очень просто и понятно. Мы просто запрашиваем у компонента `Location` текущий адрес нахождения и выводим его. В этом примере мы передали в функцию атрибут `href`, получив в результате полный адрес текущей страницы. Но нам могут понадобиться не только эти атрибуты компонента `Location` для настройки расширенного функционала. Давайте создадим еще одно простое приложение, в котором поработаем с другими атрибутами и попутно познакомимся с компонентом `Link`.

Работа с компонентом Link

Как ясно из названия, компонент `Link` служит для создания ссылок. Еще один способ создания ссылок – использование тега HTML `<a>`, входящего в состав пакета Dash HTML Components. Но если тег `<a>` больше подходит для внешних ссылок, компонент `Link` может гармонично использоваться для внутренних. Важным преимуществом этого компонента является то, что он позволяет изменить атрибут с путем без обновления страницы. Это происходит так же быстро и легко, как изменение значения любого другого интерактивного компонента. В измененном приложении мы добавим как тег `<a>`, так и пару ссылок при помощи компонента `Link`, чтобы вы увидели разницу между ними в плане обновления страницы и вообще познакомились с обоими методами на практике. В компоненте `Location` по-прежнему будет храниться ссылка на текущую страницу, но мы также извлечем и другие атрибуты объекта и посмотрим, как их можно использовать. Итак, создадим элементы для макета нашего приложения.

1. Добавим компонент `Location`:

```
dcc.Location(id='location')
```

2. Добавим компонент `<a>`, адресующий внутреннюю страницу при помощи относительного пути, воспользовавшись пакетом Dash HTML Components:

```
html.A(href='/path',
       children='Go to a directory path'),
```

3. Разместим в приложении компонент `Link`, указывающий на страницу с атрибутом поиска (параметрами запроса):

```
dcc.Link(href='/path/search?one=1&two=2',
         children='Go to search page')
```

4. Добавим еще один компонент `Link`, на этот раз указывающий на страницу с хеш-строкой, как показано ниже:

```
dcc.Link(href='path/?hello=HELLO#hash_string',
         children='Go to a page with a hash')
```

5. Создадим пустой элемент `div` для вывода информации:

```
html.Div(id='output')
```

Приложение осталось почти таким же, мы просто добавили пару ссылок. Эти ссылки будут отображаться на странице в привычном для них виде. Теперь давайте извлечем и отобразим при помощи функции обратного вызова интересующие нас части компонента `Location`.

1. Напишем декоратор функции с отдельными элементами `Input` для каждого атрибута компонента `Location`. Как вы увидите, в компоненте хранится полная ссылка на страницу, но мы можем извлечь отдельно все ее составляющие:

```
@app.callback(Output('output', 'children'),
              Input('location', 'pathname'),
              Input('location', 'search'),
              Input('location', 'href'),
              Input('location', 'hash'))
```

2. Объявим функцию обратного вызова с параметрами для каждой составляющей ссылки. Обратите внимание, что мы добавили к параметру `hash` символ подчеркивания, чтобы не возникало конфликтов с одноименной встроенной функцией Python:

```
def show_url_parts(pathname, search, href, hash_)
```

3. Возвращать будем разные атрибуты ссылки в пустом элементе `div`:

```
return html.Div([
    f"href: {href}",
    f"path: {pathname}",
    f"search: {search}",
    f"hash: {hash_}"
])
```

Запуск этого приложения приведет к отображению трех ссылок, по нажатии на которые будет меняться содержимое элемента `div` с атрибутами, как показано на рис. 11.1.

Как видите, мы использовали компонент `Link` для изменения ссылки и компонент `Location` для извлечения нужных нам атрибутов. Вы, наверное, уже представляете, как можно использовать это в более сложных функциях обратного вызова, не ограничивающихся простым выводом атрибутов. Давайте по-

смотрим, как можно разобрать адрес и извлечь параметры запроса и их значения при помощи функции `parse_qs` в Python:

```
from urllib.parse import parse_qs
parse_qs('1=one&2=two&20=twenty')
{'1': ['one'], '2': ['two'], '20': ['twenty']}
```

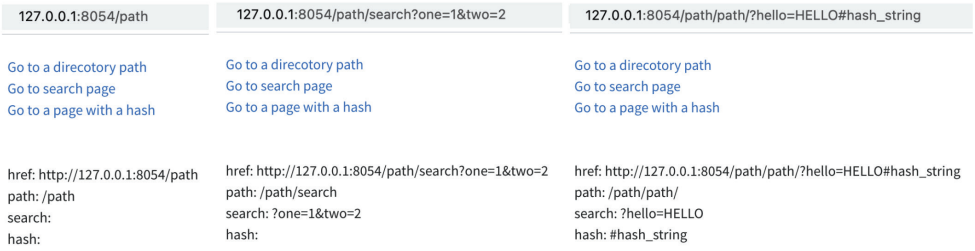


Рис. 11.1. Различные составляющие для разных ссылок

Теперь мы можем делать с этими значениями что угодно. Ниже приведен практический пример с использованием нашего набора данных – так пользователи могут делиться своими графиками с определенным набором опций при помощи простой ссылки:

```
parse_qs('country_code=CAN&year=2020&indicator=SI.DST.02ND.20')
{'country_code': ['CAN'], 'year': ['2020'], 'indicator': ['SI.DST.02ND.20']}
```

Получив название страны, год и нужный индикатор, вы можете использовать эти значения в качестве элементов ввода функции обратного вызова и строить соответствующие графики. Также можно себе представить, что пользователь, осуществляя изменения в каком-то интерактивном компоненте, попутно меняет и компонент `Location`, что облегчает задачу распространения ссылок. Очень важно еще раз подчеркнуть, что все это происходит без обновления страницы, так что никакого замедления рабочего процесса не будет.

Теперь давайте посмотрим, как это работает на практике применительно к нашему набору данных.

Разбор ссылок и использование их составляющих для изменения приложения

Познакомившись с компонентами `Location` и `Link`, мы бы хотели применить полученные знания на практике. Давайте добавим в наше приложение 169 страниц с использованием всего трех функций обратного вызова и добавлением нескольких элементов на макет. Мы предоставим пользователю возможность выбрать страну из выпадающего списка. Выбор страны приведет к изменению ссылки и внешнего вида нашего макета. В новом макете мы увидим заголовок страницы, график по стране и таблицу с данными о ней.

На рис. 11.2 показан внешний вид приложения после выбора страны.

Рис. 11.2. Страница выбранной страны с графиком и сравнением с другими странами

Как видите, у нас есть шаблон страницы для выбранной страны, который будет отображаться при указании пользователем конкретной страны. В противном случае он увидит главный экран приложения, на который всегда можно вернуться, нажав на кнопку **Home**.

Сначала давайте подумаем, как можно изменить структуру приложения.

Адаптирование приложения под множественные макеты

На данный момент у нас есть базовая структура приложения, которую мы заложили еще в главе 1. Вспомните ее, посмотрев на рис. 11.3.

Все останется без изменений, за исключением части с объявлением макета. На данный момент у нас имеется единственный макет, а все компоненты добавляются в корневой элемент `div`. Мы использовали закладки, а также компоненты `Row` и `Col` для экономии пространства и большей мобильности отображаемых элементов. Для создания новой структуры приложения нам понадобится сделать один главный макет, который будет служить основой. В этом макете будет находиться пустой элемент `div` с соответствующим контентом в зависимости от текущей ссылки. На рис. 11.4 показано, как может выглядеть основа приложения. Она никогда не будет оставаться пустой. Обратите внимание, что мы также добавили навигационную панель, на которую можно вынести некоторые элементы. Эту панель также можно рассматривать как новый компонент в нашем приложении.

Части приложения	app.py

Импорт (стандартная заготовка)	`import dash`
	`import dash_html_components as html`
	`import dash_core_components as dcc`

Создание экземпляра приложения	`app = dash.Dash(__name__)`

Макет приложения: список HTML и/или интерактивных компонентов	`app.layout = html.Div([`
	`  dcc.Dropdown()`
	`  dcc.Graph()`
	`  ...`
	`])`

Функции обратного вызова	`@app.callback()`
	`  ...`
	`@app.callback()`
	`  ...`

Запуск приложения	`if __name__ == '__main__':` `  app.run_server()`

Рис. 11.3. Структура приложения Dash

Рис. 11.4. Основной макет приложения с пустой областью страницы

Совет

Как видно на примере основного макета с пустым содержанием, вы можете использовать Dash для создания рабочих прототипов приложений во время их разработки. Перед тем как приступить к написанию кода, вы можете быстро построить нужный макет, обсудить его с инвесторами или заказчиками и только затем начинать работу. Также вы можете в процессе работы изолировать или удалять определенные элементы, чтобы аудитория лучше понимала общую структуру приложения.

Итак, давайте приступим к доработке нашего приложения, а начнем с создания нескольких отдельных макетов.

1. **Основной макет.** Этот макет будет выступать в качестве основы приложения. Он будет содержать навигационную панель и выпадающий список со странами. Здесь мы просто объявим компонент `NavbarSimple`, а подробно обсудим его далее. В этом макете также будет располагаться подвал со вкладками, который мы создали ранее. В теле макета, как и в приложении из начала главы, будет находиться компонент `Location`, а также пустой элемент `div`, в котором будет отображаться нужный макет:

```
main_layout = html.Div([
    dbc.NavbarSimple([
        ...
    ]),
    dcc.Location(id='location'),
    html.Div(id='main_content'),
    dbc.Tabs([
        ...
    ])
])
```

2. **Дашборд с индикаторами.** Это часть макета, с которой мы работали до сих пор, без изменений. Мы просто сохраняем ее в переменную и будем передавать в `div main_content` при выполнении определенного условия (если ссылка не содержит страну):

```
indicators_dashboard = html.Div([
    # Все компоненты, добавленные ранее
])
```

3. **Дашборд со страной.** Этот дашборд мы также сохраняем в переменную и отображаем, когда в ссылке есть страна. Содержимое этого макета (график и таблица) будет отражать информацию о выбранной стране:

```
country_dashboard = html.Div([
    html.H1(id='country_heading'),
    dcc.Graph(id='country_page_graph'),
    dcc.Dropdown(id='country_page_indicator_dropdown'),
    dcc.Dropdown(id='country_page_contry_dropdown'),
    html.Div(id='country_table')
])
```

4. **Проверочный макет.** Здесь у нас будет простой список в виде компонента Dash, содержащий три перечисленных макета. Суть этого списка состоит в демонстрации Dash того, какие макеты готовы для отображения в приложении. При отображении содержимого одного макета

без остальных некоторые компоненты присутствовать в приложении не будут, и обратные вызовы будут нарушены. Атрибут приложения `validation_layout` решает за нас эту проблему. Это очень простой и удобный способ управления макетами в приложении в целом, и без него не обойтись в более сложных приложениях со множеством макетов. Вы можете думать об этом макете как об оглавлении приложения:

```
app.validation_layout = html.Div([
    main_layout,
    indicators_dashboard,
    country_dashboard,
])
```

Нам нужно также объявить атрибут приложения `layout`, чтобы Dash знал, какой макет является макетом по умолчанию. Здесь все просто:

```
app.layout = main_layout
```

Посмотрим, как можно управлять содержимым, которое будет отображаться в макете `main_layout`.

Отображение содержимого на основе ссылки

Настроив расположение макетов, нам понадобится написать простую функцию для управления их содержимым.

Функция должна проверять, содержит ли атрибут пути компонента `Location` одну из допустимых стран. Если да, будет возвращен макет `country_dashboard`. В противном случае – макет `indicators_layout`. Обратите внимание, что второе условие включает в себя все, кроме вхождения в имеющийся список стран. Поскольку в нашем приложении нет никакого другого функционала и мы не отлавливаем ошибки в ссылках, будет хорошо, если все неподходящие ссылки будут вести на главную страницу. В более сложных приложениях можно создать отдельную страницу с выводом сообщений об ошибках.

Также здесь необходимо сделать два замечания. Во-первых, атрибут `pathname` возвращает путь в виде `"/<country_name>"`, так что нам нужно извлекать его содержимое без первого символа. Во-вторых, при наличии в ссылке специальных символов они автоматически будут переводиться в кодировку URL. Декодировать эти символы можно при помощи функции `unquote`, как показано ниже:

```
from urllib.parse import unquote
unquote('Bosnia%20and%20Herzegovina')
'Bosnia and Herzegovina'
```

Пробелы в названии выбранной страны автоматически будут преобразованы в `%20`, так что без функции `unquote`, позволяющей выполнить обратное преобразование, нам не обойтись.

Ниже представлен код выделения списка доступных стран и простая функция обратного вызова, управляющая содержимым страницы на основе ссылки:

```
countries = poverty[poverty['is_country']]['Country Name'].drop_duplicates().
sort_values().tolist()

@app.callback(Output('main_content', 'children'),
              Input('location', 'pathname'))
def display_content(pathname):
    if unquote(pathname[1:]) in countries:
        return country_dashboard
    else:
        return indicators_dashboard
```

Итак, мы описали верхнеуровневую структуру приложения, и теперь нам нужно поговорить о деталях создания навигационной панели, выпадающего списка и ссылок.

Добавление динамически сгенерированных ссылок в приложение

Давайте завершим создание основного макета приложения, добавив в него навигационную панель, ссылку для перехода на главную страницу и выпадающий список со странами. Пришло время познакомиться с компонентом `NavbarSimple` из пакета Dash Bootstrap Components.

Компонент `NavbarSimple` принимает на вход несколько элементов, которые мы рассмотрим далее.

1. Создадим навигационную панель и передадим на вход аргументы `brand` и `brand_href`, указывающие текст и адрес ссылки, располагающейся на панели:

```
import dash_bootstrap_components as dbc
dbc.NavbarSimple([
    ...
], brand="Home", brand_href="/")
```

2. В качестве аргумента `children` мы передадим компонент `dbc.DropdownMenu` с подписью в виде атрибута `label`, чтобы пользователи понимали, что выбирают в списке. Аргумент `children` выпадающего списка мы заполним на следующем шаге:

```
dbc.DropdownMenu(children=[
    menu_item_1,
    menu_item_2,
    ...
], label="Select country")
```

3. Нам необходимо передать на вход `dbc.DropdownMenu` список компонентов `dbc.DropdownMenuItem`. Каждый из этих компонентов будет принимать

аргументы `children` и `href`, причем мы в обоих случаях будем передавать страну из нашего списка стран, который мы создали ранее:

```
dbc.DropdownMenu([
    dbc.DropdownMenuItem(country, href=country)
    for country in countries
])
```

В итоге мы получим следующий код создания навигационной панели:

```
dbc.NavbarSimple([
    dbc.DropdownMenu([
        dbc.DropdownMenuItem(country, href=country)
        for country in countries
    ], label='Select country')
], brand='Home',brand_href='/')
```

В результате мы создали навигационную панель нашего приложения. С таким же успехом мы могли бы добавить в этот макет вкладки с помощью компонента `Tabs`, с которым встречались в главе 1. С такой структурой вам будет очень легко добавлять любые компоненты на панель и по-своему выстраивать ее макет.

Обратите внимание, что в атрибуте `children` навигационной панели могут содержаться ссылки, которые ведут себя так же, как компонент `Link`. Так что его вы также можете использовать при необходимости.

Теперь, когда основной макет приложения подготовлен, а на страницу загружается нужный макет в зависимости от текущей ссылки, мы готовы к реализации двух других функций обратного вызова, которые помогут в создании макета `country_dashboard`.

Но тут такая ситуация. Я – ваш бывший коллега – написал код и создал весь функционал, после чего ушел из компании без объяснений, и вы никак не можете со мной связаться. Теперь вам необходимо определяться со структурой самому и вносить изменения в мой код.

Это довольно типичная ситуация, с которой вы можете сталкиваться в компаниях. Давайте посмотрим, как быть в таких случаях.

Внедрение в приложение интерактивности на основе ссылок

Создав выпадающий список, автоматически меняющий ссылку на основе выбора пользователя, мы тем самым позволили ему перемещаться со страницы на страницу. Теперь нам нужно озаботиться показом нужного содержания страницы в зависимости от выбора пользователя.

Поскольку код уже написан, мы можем запустить его в режиме отладки и рассмотреть диаграмму с текущими компонентами и их взаимосвязями.

На рис. 11.5 показан граф обратных вызовов, созданных на данный момент. Давайте используем его для понимания нынешнего функционала приложения.

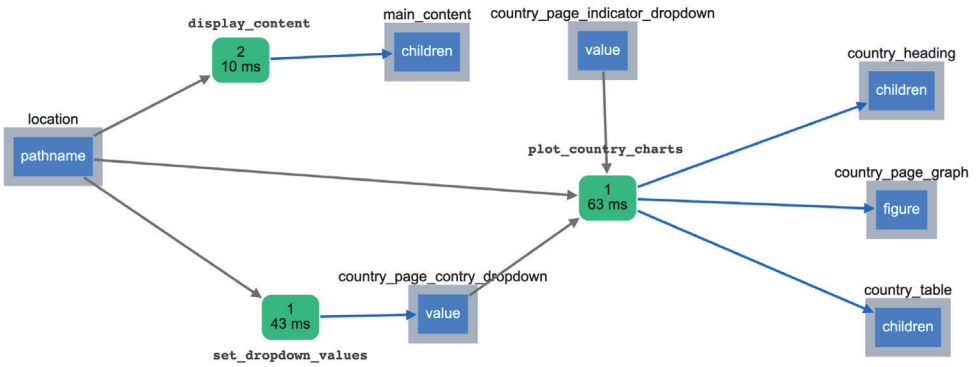


Рис. 11.5. Различные компоненты и их взаимосвязи на основе обратных вызовов

Давайте пройдемся по графу слева направо и посмотрим, что здесь происходит. Вы можете обращаться к рис. 11.2 за визуальным представлением приложения:

- все начинается с изменения атрибута `pathname` компонента `Location`. Это может произойти в результате выбора пользователем страны из списка, ручного ввода ссылки, содержащей нужную страну, или щелчка по ссылке на странице;
- изменение ссылки оказывает влияние на три компонента, что видно на рис. 11.5. Самое важное, что оно определяет содержимое атрибута `children` элемента `div` с идентификатором `main_content`. Если ссылка содержит корректную страну, будет отображен макет `country_dashboard` с помощью функции `display_content`. В результате мы увидим новые компоненты, управляемые своими функциями обратного вызова;
- если предположить, что пользователь выбрал страну корректно, отработает вторая функция `set_dropdown_values`, которая на основании указанной страны заполнит выпадающий список стран для отрисовки на графике;
- функция `plot_country_charts` принимает на вход три элемента ввода и оказывает влияние на три элемента вывода, что видно на рис. 11.5. Компонент `country_heading` – это просто заголовок первого уровня с текстом `"<country name> Poverty Data"`. Компонент `country_page_graph` представляет собой основной график на странице, а `country_table` – таблицу `dbc.Table` с информацией о выбранной стране, извлеченной из файла CSV.

После проведения такого анализа вы открываете код из репозитория, чтобы посмотреть, как этот функционал реализован на практике, и видите две функции обратного вызова, которые при необходимости можно изменить:

```
@app.callback(Output('country_page_contry_dropdown', 'value'),
              Input('location', 'pathname'))
def set_dropdown_values(pathname):
    if unquote(pathname[1:]) in countries:
```

```
        country = unquote(pathname[1:])
        return [country]

@app.callback(Output('country_heading', 'children'),
              Output('country_page_graph', 'figure'),
              Output('country_table', 'children'),
              Input('location', 'pathname'),
              Input('country_page_contry_dropdown', 'value'),
              Input('country_page_indicator_dropdown', 'value'))
def plot_country_charts(pathname, countries, indicator):
    if (not countries) or (not indicator):
        raise PreventUpdate
    if unquote(pathname[1:]) in countries:
        country = unquote(pathname[1:])
    df = poverty[poverty['is_country'] & poverty['Country Name'].
isin(countries)]
    fig = px.line(df,
                  x='year',
                  y=indicator,
                  title='<b>' + indicator + '</b><br>' + ', '.join(countries),
                  color='Country Name')
    fig.layout.paper_bgcolor = '#E5ECF6'
    table = country_df[country_df['Short Name'] == countries[0]].T.reset_index()
    if table.shape[1] == 2:
        table.columns = [countries[0] + ' Info', '']
        table = dbc.Table.from_dataframe(table)
    else:
        table = html.Div()
    return country + ' Poverty Data', fig, table
```

К этому моменту вам не составит никакого труда разобраться в коде, даже если некоторые компоненты вы видите впервые в жизни. Мы уже столько раз разбирали и изменяли структуру подобных приложений, что вы легко справитесь с этой задачей.

Это последний пример кода в данной книге и последний функционал, который мы добавили в наше приложение. Но это не значит, что приложение готово. Вы можете внести в него еще много изменений. Например, вы могли бы создать отдельные ссылки для индикаторов и выделить для них отдельную страницу по примеру со странами. Однако это может быть не так просто, поскольку некоторые индикаторы исчисляются в процентах, а другие – в абсолютных величинах. Кроме того, какие-то индикаторы стоит объединить в группу, например те, что показывают различные квинтили по численности населения. В общем, здесь у вас будет безграничный простор для творчества.

Давайте подытожим все, что мы узнали из этой главы.

Заключение

В данной главе мы познакомились с двумя очень важными компонентами, позволяющими работать со ссылками, – `Location` и `Link`. Мы создали два простых приложения, на примере которых научились извлекать составляющие части ссылок и поэкспериментировали с разными подходами к их использованию.

Также мы узнали, как можно модифицировать структуру приложения с помощью разбора ссылок. Мы создали основной макет приложения с пустым элементом `div`, содержание которого определяется на основе ссылок.

После этого мы внедрили новый функционал в наше приложение и даже побывали в шкуре разработчика, которого нерадивый коллега бросил разбираться с приложением в одиночку.

Теперь, когда мы познакомились с основными составляющими приложения, включая макеты, компоненты и ссылки, мы можем приступить к развертыванию своего приложения на сервере, чтобы им могли пользоваться наши друзья и коллеги из любой точки планеты.

Об этом мы и поговорим в следующей главе.

Глава 12

Развертывание приложения

Мы проделали большую работу, и я вижу по глазам, что вам не терпится поделиться ей со всем миром. В этой главе вы получите такую возможность, ведь мы сами настроим сервер и выполним развертывание приложения на публичном адресе.

По своей сути наши действия будут сводиться к переносу приложения и данных на другой компьютер и запуску, похожему на запуск на локальном компьютере. Но для того чтобы приложение было доступно всем пользователям глобальной сети, необходимо настроить аккаунт, сервер и интерфейс **Web Server Gateway Interface** (WSGI).

В процессе мы кратко коснемся системы контроля версий файлов исходного кода **Git** и выполним базовые системные настройки в **Linux**. Мы не будем особенно вдаваться в подробности и сделаем минимум необходимого для развертывания нашего приложения. Все тонкости и нюансы используемых в этой главе инструментов вы сможете освоить самостоятельно или с помощью специализированной литературы. Подход, которого мы будем придерживаться, позволит нам в минимальные сроки и с минимальными усилиями опубликовать приложение в интернете и сделать его доступным для всех пользователей. Причем мы будем пользоваться не какими-то упрощенными инструментами. Наоборот, мы будем использовать полноценные инструменты, но с простыми базовыми настройками. Это позволит нам не усложнять все поначалу, но даст возможность произвести более тонкие настройки по мере накопления знаний.

Также мы рассмотрим альтернативный способ развертывания приложений при помощи Dash Enterprise – платной платформы Dash, которая подойдет для крупных организаций.

Темы, которые будут рассмотрены в главе:

- основы рабочего процесса разработки, развертывания и обновления приложений;
- аренда виртуального сервера и настройка аккаунта;
- подключение к серверу при помощи **Secure Shell (SSH)**;
- запуск приложения на сервере;
- настройка и запуск приложения через WSGI-сервер;
- настройка и конфигурирование веб-сервера;
- поддержка приложения и его обновление;
- развертывание и масштабирование приложений Dash с помощью Dash Enterprise.

Технические требования

Нам понадобится сервер Linux, подключенный к интернету, файлы с данными и наш код. Мы установим **Gunicorn** (Green Unicorn, WSGI-сервер) и **nginx** (веб-сервер), а также пакеты Python для нашего приложения: Dash и его основные библиотеки, Dash Bootstrap Components, pandas и sklearn. Кроме того, нам понадобится аккаунт в системе контроля версий **Git**, и в этой главе мы будем использовать GitHub.

На протяжении книги мы сначала проверяли новый функционал в JupyterLab, после чего внедряли его в наше приложение. И этот подход не претерпит изменений. Мы просто добавим несколько шагов и компонентов для развертывания после внесения изменений. Итак, давайте приступим к описанию рабочего процесса.

Основы рабочего процесса разработки, развертывания и обновления приложений

Когда мы говорим о развертывании приложения, это автоматически означает, что мы довольны тем, что у нас получилось, и наш код и данные готовы к путешествию на сервер. В этой главе мы сосредоточим внимание на настройке требуемой инфраструктуры для запуска нашего приложения онлайн. Мы не будем сильно углубляться в тему, а выполним простую настройку с минимальными требованиями. В качестве провайдера мы будем использовать Linode. Мы выбрали этот хостинг по причине того, что в нем реализована философия *открытого облака* (open cloud). Это означает, что мы будем работать с «чистым» сервером Linux, использующим компоненты и пакеты с открытым исходным кодом, позволяющие выполнять любые настройки и миграции. Потенциальный вызов в этом случае кроется в том, что с большой свободой приходит большая ответственность и сложность. В главе 4 мы уже обсуждали компромиссы, на которые приходится идти в процессе выбора высокоуровневого или низкоуровневого программного обеспечения, и в данном случае мы будем работать со вторым вариантом для запуска нашего приложения.

Если у вас есть опыт запуска и настройки серверов, вы можете все сделать самостоятельно и пропустить большую часть этой главы. Новичков же я обрадую тем, что хоть вы и будете все делать сами, мы пройдем по всем шагам и выполним простейшую настройку необходимой для запуска приложения инфраструктуры. Это позволит вам с легкостью выполнить развертывание своего приложения и постепенно изучить варианты настройки окружения, с которым вы будете работать.

Процесс развертывания приложения условно можно разделить на следующие три фазы:

- **локальная рабочая машина:** об этом мы говорили на протяжении всей книги, и вы должны уже неплохо ориентироваться в локальной инфраструктуре;
- **система контроля версий файлов исходного кода:** для запуска вашего приложения на сервере система контроля версий вам может и не

понадобиться, но с развитием приложения и привлечения к его разработке нескольких человек без нее вам будет не обойтись;

- **сервер с требуемой инфраструктурой и настройкой:** именно здесь будет выполняться ваше приложение.

На рис. 12.1 схематически показаны эти три элемента, и далее мы поговорим о них и их месте в жизненном цикле разработки и развертывания приложений.

Рис. 12.1. Три главных компонента цикла разработки, развертывания и обновления приложения

Один или несколько разработчиков пишут код приложения на своих компьютерах, который, в свою очередь, мигрирует между локальными машинами и центральным *репозиторием Git* (Git repository). Обратите внимание, что исходный код не пересылается разработчиками друг другу напрямую (например, по электронной почте) – вместо этого все изменения/дополнения накапливаются в репозитории Git. Преимуществ у архитектуры с центральным хранилищем Git великое множество. Это большая тема для обсуждения, по которой написаны отдельные книги. Мы же расскажем только о базовых принципах общих репозиториев, а более глубоко в эту тему вы сможете при желании и необходимости погрузиться самостоятельно. Важнейшим плюсом использования центрального репозитория является возможность совместной работы с единым набором изменений, что облегчает деятельность разработчиков. За репозиторием должны быть закреплены один или несколько администраторов и сотрудники, которые решают, что будет входить в состав репозитория, а что нет. Также им предстоит разрешать конфликтные ситуации. К примеру, два или более разработчиков могут одновременно работать с одним и тем же исходным файлом, в результате чего могут возникать конфликты изменений. И должен быть кто-то, способный решить, что необходимо делать в таких ситуациях.

Утвержденные версии кода отправляются из репозитория на сервер для публикации. Как видите, код на данной стадии мигрирует только в одном направлении. Даже если вы работаете над приложением один, мы рекомендуем использовать репозиторий Git для управления изменениями.

Еще одним важным преимуществом применения центрального репозитория является то, что вместе со всеми изменениями, называемыми *коммитами* (commit), хранятся метаданные о том, кем и когда было выполнено то или иное изменение. Кроме того, вы можете посмотреть, к какой *ветви* (branch) изменений относится конкретное действие. Если вам необходимо выполнить *откат*

(roll back) изменения, вы можете *извлечь* (check out) определенный коммит или ветвь и вернуть репозиторий к определенному состоянию. Пока приложение работает, вы можете внести необходимые изменения, после чего снова отправить их в хранилище.

Мы очень поверхностно описали принципы работы репозитория Git, но на данном этапе этого будет вполне достаточно. Мы будем использовать Git для развертывания приложения и в процессе покажем основные команды. В заключительном разделе главы мы также рассмотрим процесс внесения изменений в один из исходных файлов, чтобы продемонстрировать, как изменения попадают в рабочее приложение.

На данном этапе у вас есть код приложения, исправно работающий на локальной машине, и все необходимые файлы с данными, и вам нужно отправить все исходники в репозиторий Git, а затем – на сервер для развертывания. Также вы могли бы просто клонировать репозиторий этой книги и развернуть его на вашем сервере.

Теперь, когда мы обсудили рабочий процесс в целом и коснулись отдельных его компонентов, можно приступать к реальной работе, и начать необходимо с создания аккаунта на хостинге Linode.

Аренда виртуального сервера и настройка аккаунта

Зарегистрироваться на сайте хостинга Linode можно очень просто, вы можете сделать это, перейдя по ссылке **https://login.linode.com/signup**. После завершения процесса регистрации и предоставления платежной информации вы можете создать собственный *Linode* (Linux + node) – виртуальный сервер со своим IP-адресом.

Это довольно простой процесс, который выполняется прямо с главной страницы сайта. На рис. 12.2 показаны некоторые опции, доступные при создании и управлении вашим аккаунтом.

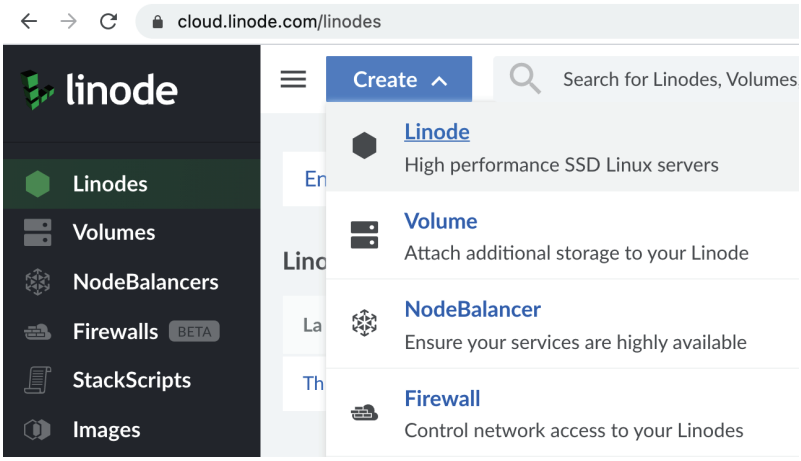


Рис. 12.2. Объекты, которые вы можете создать на хостинге Linode

Нажав на пункт меню **Create** и выбрав вариант **Linode**, вы будете перемещены на страницу с разными вариантами установки. Переключитесь на вкладку **Distributions**. Здесь вам будет предложен выбор из нескольких дистрибутивов программного обеспечения на базе Linux, но с использованием разных компонентов. Каждый дистрибутив лучше подходит для каких-то своих целей. Мы для нашего узла Linode выберем дистрибутив Ubuntu. Вы можете посмотреть на содержимое других вкладок. Например, на вкладке **Marketplace** можно создать полноценный установщик для популярных программных пакетов всего за пару щелчков. На рис. 12.3 показан выбранный нами дистрибутив. После осуществления выбора вам необходимо будет указать план использования услуги. Вы можете начать с минимального, а затем расширить его при необходимости. На рис. 12.3 видно, что мы остановили свой выбор на дистрибутиве Ubuntu.

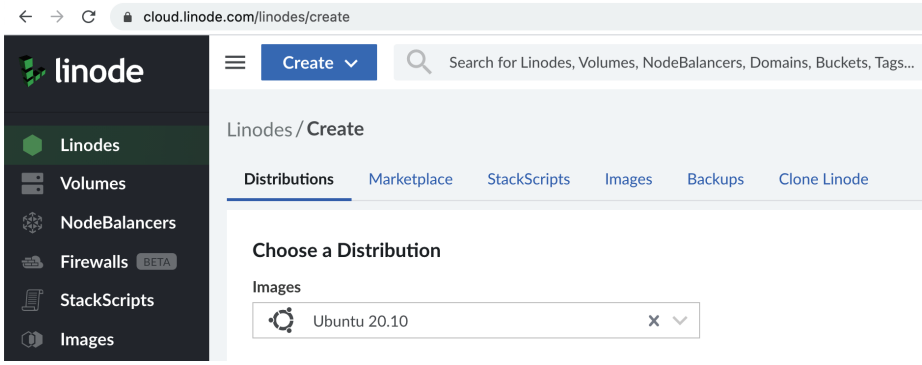


Рис. 12.3. Выбор дистрибутива для узла

Выполнив все необходимые настройки, нажмите на кнопку **Create**, что перенесет вас на страницу с созданным узлом Linode. На рис. 12.4 показана верхняя область этой страницы.

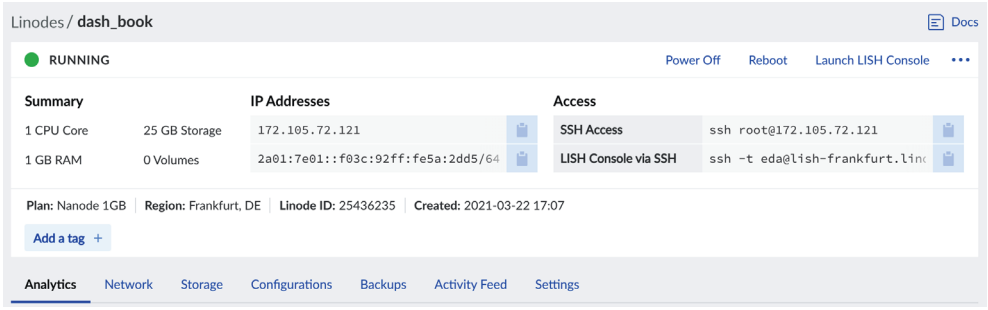


Рис. 12.4. Страница созданного узла Linode

Здесь вы увидите всю сопутствующую информацию касательно вашего узла, к которой вы всегда можете вернуться, если вам понадобится посмотреть какие-то данные по эффективности сервера или изменить его конфигурацию. Впоследствии вам может потребоваться изменить план пользования услугами провайдера или расширить хранилище.

В данный момент нас больше всего интересует раздел **SSH Access**. Сейчас мы будем использовать информацию из этого раздела для подключения к нашему серверу, причем не из веб-интерфейса.

> **Важно**
>
> Linode предлагает несколько способов взаимодействия с вашим аккаунтом. Среди прочих выделяются доступ посредством *интерфейса программирования приложений* (application programming interface, API) и с помощью *интерфейса командной строки* (command-line interface, CLI). Эти инструменты позволяют получить полный доступ к вашим ресурсам и выполнять любые действия, доступные из веб-интерфейса. Для простых задач бывает удобнее использовать веб-интерфейс, но когда необходимо автоматизировать и масштабировать процессы, API и CLI становятся просто незаменимыми.

Теперь, когда мы арендовали и запустили сервер, можно начать работать с ним. И делать мы это будем посредством интерфейса SSH.

Подключение к серверу при помощи Secure Shell (SSH)

SSH представляет собой протокол для безопасного перемещения данных в рамках небезопасной сети. С помощью этого протокола можно получить доступ к нашему серверу и запустить код из командной строки с использованием терминала на локальной машине.

Скопируйте команду `ssh root@127.105.72.121`, воспользовавшись специальной иконкой справа от поля **SSH Access**. Теперь откройте приложение с терминалом на локальном компьютере, введите скопированную команду и нажмите на **Enter**:

```
ssh root@172.105.72.121
The authenticity of host '172.105.72.121 (172.105.72.121)' can't be established.
ECDSA key fingerprint is SHA256:7TvPpP9fko2gTGG1lW/4ZJC+jj6fB/nzVzlW5pjepyU.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '172.105.72.121' (ECDSA) to the list of known hosts.
root@172.105.72.121's password:
```

Как видите, в процессе подключения нам задали два вопроса: сначала узнали, хотим ли мы подключиться и добавить IP-адрес в список знакомых хостов, на что мы ответили `yes`, а затем запросили пароль.

После ввода пароля, который вы заранее должны были создать для своего узла (а не для аккаунта на сайте), появится строка приглашения следующего вида:

```
root@localhost:~#
```

Итак, мы получили root-доступ к серверу из терминала. Мы сразу же советуем вам обновить системные пакеты, введя команды, показанные ниже:

```
apt-get update
apt-get upgrade
```

Позже вам может понадобиться обновить пакеты, которые вы будете устанавливать отдельно.

Получив root-доступ к серверу, вы будете наделены очень большими полномочиями, что может поставить под угрозу безопасность сервера. Если кто-то другой сможет получить доступ к вашему серверу от имени привилегированного пользователя root, он без труда сможет добавлять и удалять какие угодно файлы – без всяких ограничений.

В связи с этим мы рекомендуем вам создать пользователя с ограниченными правами и подключаться к серверу от его имени. Для этого нужно сделать следующее.

1. Создайте пользователя с любым именем:

   ```
   adduser elias
   ```

2. Далее вам будет необходимо дважды ввести пароль для создаваемого пользователя. На этот момент у вас будет уже три пароля: один для аккаунта на сайте, второй для пользователя root и третий – для созданного пользователя с ограниченными правами. При создании нового пользователя вы также по желанию можете ввести информацию о нем, которую у вас тоже попросят. Если не хотите, просто нажимайте на клавишу **Enter** и идите дальше. Несмотря на ограниченный статус созданного пользователя, мы можем добавить его в специальную привилегированную группу `sudo` (superuser do!). Это позволит пользователю временно получать привилегии root-пользователя, когда нам необходимо выполнить какие-то административные задачи или получить доступ к конфиденциальным файлам. Добавление пользователя в эту группу выполняется следующим образом:

   ```
   adduser elias sudo
   ```

3. Теперь, когда мы создали пользователя и временно выдали ему членство в группе `sudo`, мы можем отключиться от пользователя root, чтобы затем подключиться от имени elias:

   ```
   exit
   ```

4. Появится сообщение о завершении подключения к серверу и приглашение к работе с локальной машиной. Подключимся к серверу под новым пользователем, как показано ниже:

   ```
   ssh elias@172.105.72.121
   ```

5. Вы увидите строку приглашения к работе от имени созданного пользователя:

```
elias@localhost:~$
```

Итак, мы готовы к работе с нашим приложением на сервере, но сначала давайте посмотрим, как можно выполнять привилегированные команды, используя команду `sudo`.

1. Давайте попробуем получить доступ к одному из файлов системного лога, например с использованием команды `cat`, как показано ниже:

```
elias@localhost:~$ cat /var/log/syslog
cat: /var/log/syslog: Permission denied
```

2. Как и следовало ожидать, доступ мы не получили. Теперь давайте предварим нашу инструкцию командой `sudo`:

```
elias@localhost:~$ sudo cat /var/log/syslog
[sudo] password for elias:
```

3. Введите пароль для созданного пользователя, и вы получите доступ к требуемым файлам.

Вам зачастую будет требоваться выполнить какие-то действия в привилегированном режиме, и для предоставления вам временного доступа к правам root вы можете пользоваться командой `sudo`.

На данный момент наш сервер подготовлен к работе, но нам еще нужно перенести на него наши скрипты и файлы с данными, а также установить необходимые пакеты Python.

Запуск приложения на сервере

В этом разделе мы будем делать в точности то, что делали в главе 1, – клонируем данные и код из репозитория GitHub и поместим их на сервер, после чего установим нужные пакеты и попытаемся запустить приложение.

Обычно на таких серверах Python бывает уже установлен, но никогда не будет лишним проверить, как с этим обстоят дела. Самый простой способ сделать это – запустить в командной строке команду `python --version`. Обратите внимание, что команда `python` может означать интерпретатор Python 2. Для проверки установки третьей версии вы можете воспользоваться командой `python3`. Это же относится и к установщику пакетов `pip`, вместо которого можно пользоваться командой `pip3`.

Выполнив команду `python3 --version`, я получил версию `3.8.6`. К моменту чтения вами этой книги версия по умолчанию может измениться. К тому же во время написания книги версия Python 3.9 была уже выпущена и считалась стабильной. Вот какое сообщение я получил, попытавшись запустить эту версию из командной строки:

```
elias@localhost:~$ python3.9
Command 'python3.9' not found, but can be installed with:
sudo apt install python3.9
```

Нам лишний раз напомнили, что для установки Python нам потребуется воспользоваться привилегированными правами группы sudo. Мы рассмотрели несколько примеров использования команд Linux, но администрирование этой операционной системой – это отдельная большая тема, которую вы можете освоить самостоятельно.

Теперь давайте активируем *виртуальное окружение* (virtual environment), как мы уже делали это в главе 1.

1. Создайте виртуальное окружение в папке dash_project (или любой другой по вашему желанию). Это также приведет к созданию самой папки с выбранным именем. Учтите, что для этого вам может понадобиться установить модуль venv, и соответствующую команду для этого вам подскажет операционная система при попытке выполнения следующей команды:

   ```
   python3 -m venv dash_project
   ```

2. Активируйте виртуальное окружение. После этого вы должны видеть название окружения в круглых скобках перед именем пользователя, как показано ниже:

   ```
   source dash_project/bin/activate
   (dash_project) elias@localhost:~/dash_project$
   ```

3. Перейдите в папку с помощью команды cd:

   ```
   cd dash_project
   ```

4. Теперь нам необходимо клонировать репозиторий GitHub и перенести все наши исходные файлы и файлы с данными на сервер. Мы в качестве примера будем использовать репозиторий для этой книги, но я рекомендую вам попробовать запустить и клонировать собственный репозиторий. Выполните следующую команду:

   ```
   git clone https://github.com/PacktPublishing/Interactive-
   Dashboards-and-Data-Apps-with-Plotly-and-Dash
   ```

5. Теперь нам необходимо установить необходимые пакеты. Для этого перейдем в главную папку проекта и выполним следующую команду:

   ```
   cd Interactive-Dashboards-and-Data-Apps-with-Plotly-and-Dash/
   pip install -r requirements.txt
   ```

6. Далее мы можем перейти в папку любой главы книги и запустить соответствующее приложение. Давайте попробуем запустить приложение из главы 11 следующим образом:

   ```
   cd chapter_11
   python app_v11_1.py
   ```

Проделав все эти действия, мы пришли к такому же результату, как и раньше, когда запускали приложение на локальном компьютере, что видно по рис. 12.5.

```
Dash is running on http://127.0.0.1:8050/

 * Serving Flask app "app_v11_1" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
```

Рис. 12.5. Приложение, запущенное на сервере

Если вы получили такой же результат, значит, ваше приложение успешно запустилось на сервере. В то же время нас не может не смущать предупреждение, отмеченное красным, о том, что приложение работает на сервере в режиме разработки и использовать это окружение в качестве рабочего не рекомендуется.

Что ж, придется устанавливать веб-сервер. Также нам нужно будет использовать интерфейс, облегчающий взаимодействие нашего веб-фреймворка (Flask) с любым выбранным нами веб-сервером. Этот интерфейс называется **WSGI** (произносится как уиз-ги или просто виски!).

Но давайте для начала разберемся во всех компонентах и фазах при работе с нашим приложением. На рис. 12.6 показана диаграмма с последовательностью запросов и ответов, возникающих при доступе пользователя к нашему приложению из браузера.

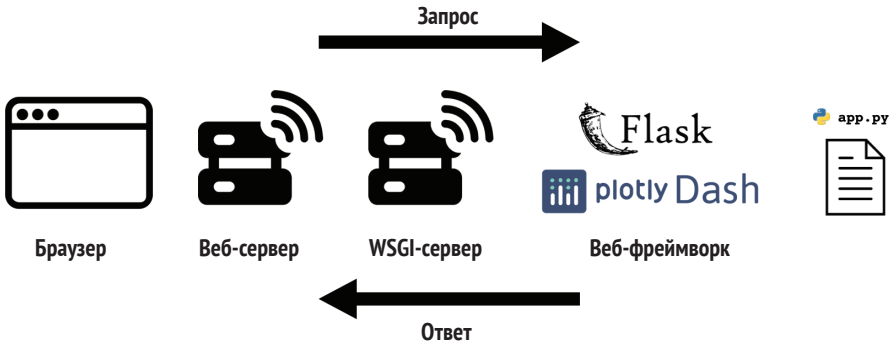


Рис. 12.6. Компоненты приложения на публичном сервере

Запрос идет слева направо вплоть до достижения фреймворка Dash, который запускает модуль `app.py`. После этого код нашего приложения генерирует ответ, проходящий через те же компоненты в обратном направлении, пока не достигнет браузера.

Давайте кратко разберем все элементы, представленные на рис. 12.6.

- **Браузер:** здесь все просто, это может быть любой клиент, понимающий *протокол гипертекстовой передачи* (HyperText Transfer Protocol, HTTP). При вводе пользователем *унифицированного указателя ресурса* (Uniform Resource Locator, URL) и нажатия на клавишу **Enter** браузер посылает соответствующий запрос веб-серверу.

- **Веб-сервер:** работа веб-сервера заключается в обработке поступающих запросов. Проблема состоит в том, что наш сервер не исполняет код на Python, так что нам нужно как-то получать запросы, интерпретировать их и возвращать ответ.
- **WSGI-сервер:** это программное обеспечение промежуточного уровня, говорящее одновременно и на языке сервера, и на Python. В его присутствии веб-фреймворку (в нашем случае Flask) не нужно будет заботиться об обработке множества запросов. Вместо этого он сможет сконцентрироваться на создании веб-приложения, а из обязательных требований останется только соответствие спецификации WSGI. Это также означает, что мы сможем менять наш веб-сервер и/или WSGI-сервер, не внося никаких изменений в код приложения.
- **Веб-фреймворк:** это веб-фреймворк Flask, на основе которого построен Dash. Приложение Dash по сути является приложением Flask, о чем мы уже говорили ранее.

На данном этапе нам не нужно знать о веб-серверах и WSGI-серверах больше, чем мы уже знаем. Давайте посмотрим, как просто можно запустить приложение с использованием WSGI-сервера.

Настройка и запуск приложения через WSGI-сервер

Мы уже запускали наше приложение с помощью команды `python app.py`. Также при использовании библиотеки `jupyter_dash` мы пользовались для запуска методом `app.run_server`. Сейчас мы осуществим запуск приложения при помощи WSGI-сервера **Gunicorn**.

Это можно сделать с использованием команды, шаблон которой представлен ниже:

```
gunicorn <app_module_name:server_name>
```

Здесь есть два важных отличия от предыдущих способов. Во-первых, мы используем только имя модуля или файла без указания расширения `.py`. После этого через двоеточие следует имя сервера. Это просто переменная, которую необходимо объявить, и это можно сделать с помощью одной строки кода, которую нужно вставить после объявления верхнеуровневой переменной `app`, как показано ниже:

```
app = dash.Dash(__name__)
server = app.server
```

Теперь, когда мы определили наш сервер как `server` в рамках файла `app.py`, можно запустить приложение из командной строки следующим образом:

```
gunicorn app:server
```

Вот и вся работа с WSGI-сервером!

После выполнения изменений в исходном файле можно перейти в папку с нашим приложением и запустить его, используя команду, показанную выше. На рис. 12.7 показан вывод этой команды.

```
[2021-03-23 14:50:51 +0000] [54222] [INFO] Starting gunicorn 20.0.4
[2021-03-23 14:50:51 +0000] [54222] [INFO] Listening at: http://127.0.0.1:8000 (54222)
[2021-03-23 14:50:51 +0000] [54222] [INFO] Using worker: sync
[2021-03-23 14:50:51 +0000] [54224] [INFO] Booting worker with pid: 54224
```

Рис. 12.7. Запуск приложения с использованием WSGI-сервера Gunicorn

Как видите, приложение работает нормально. Также мы видим, что оно запустилось на другом порту. По умолчанию приложения Dash запускаются с использованием порта 8050, а здесь присутствует порт 8000.

Итак, мы сделали еще один шаг навстречу браузеру и пользователю. Кажется, что приложение с использованием WSGI-сервера работает нормально. Так давайте настроим веб-сервер, чтобы наше приложение стало доступно всем.

Настройка и конфигурирование веб-сервера

В нашем примере мы будем использовать веб-сервер ***nginx***. Вы можете остановить ваше приложение, нажав сочетание клавиш **Ctrl**+**C**. Также можно сделать это, используя команду `kill`, если вам известен идентификатор соответствующего процесса.

Вы всегда можете выполнить команду определения статусов процессов `ps -A` из командной строки, чтобы увидеть все запущенные процессы. Далее вы можете прокрутить список вручную в поисках процесса `gunicorn`, а можете через вертикальную черту задать поиск нужной вам строки следующим образом:

```
ps -A | grep gunicorn
```

Запуск этой команды при работающем приложении привел к выводу, показанному на рис. 12.8.

```
elias@localhost:~$ ps -A | grep gunicorn
  54222 pts/1    00:00:00 gunicorn
  54224 pts/1    00:00:02 gunicorn
```

Рис. 12.8. Поиск процесса по определенному шаблону

Идентификаторы процессов оказались такими же, как мы видели при запуске Gunicorn. Для остановки процесса воспользуйтесь командой `kill` следующим образом:

```
kill -9 54222
```

Теперь, когда мы убедились, что наше приложение успешно запускается с использованием WSGI-сервера, пришло время настроить веб-сервер.

Как уже было сказано раньше, несмотря на использование простейших настроек с целью облегчения примеров, мы будем использовать один из самых мощных веб-серверов.

Давайте начнем с его установки. Запустите в командной строке следующую команду:

```
sudo apt install nginx
```

Теперь нам нужно создать конфигурационный файл для нашего приложения. В процессе установки nginx, помимо прочего, была создана папка `sites-enabled`. Нам нужно создать в ней файл конфигурации с базовыми настройками. Мы можем использовать для этого любой текстовый редактор, и простейшим из редакторов в операционной системе Linux является `nano`. Запуск этой команды с указанием имени файла приведет к его открытию на редактирование (или созданию в случае его отсутствия).

Из командной строки запустите команду, показанную ниже, для открытия файла:

```
sudo nano /etc/nginx/sites-enabled/dash_app
```

Откроется пустой файл, в который вы можете скопировать конфигурационную информацию, показанную ниже, не забыв изменить IP-адрес после слова `server_name` на собственный:

```
server {
    listen 80;
    server_name 172.105.72.121;
    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

Этот код содержит конфигурацию контекста `server`. В частности, он указывает ему на то, что необходимо прослушивать порт `80`, являющийся портом по умолчанию для веб-серверов. Он также определяет `server_name` как внешний IP-адрес сервера. Позже вы можете использовать это для определения вашего собственного доменного имени.

Затем в блоке `location /` определяется поведение сервера. Наиболее важным для нас является то, что мы указываем nginx в качестве прокси-сервера с помощью инструкции `proxy_pass` и говорим ему слушать адрес и порт, который слушает Gunicorn. Таким образом, цикл замкнулся. Наш веб-сервер будет посылать запросы и получать ответы через корректный адрес и порт, где интерфейс с кодом на Python управляется при помощи Gunicorn.

Во время установки nginx создается файл конфигурации по умолчанию, который нам необходимо отвязать при помощи следующей команды:

```
sudo unlink /etc/nginx/sites-enabled/default
```

После внесения этого изменения нужно перезагрузить nginx. Не забывайте об этом и в следующие разы, когда будете делать какие-либо изменения. Перезагрузку nginx можно выполнить, запустив следующую команду:

```
sudo nginx -s reload
```

Теперь можно снова запустить наше приложение командой `gunicorn app:server`, но на этот раз открыть его с использованием браузера и нашего внешнего IP-адреса, как показано на рис. 12.9.

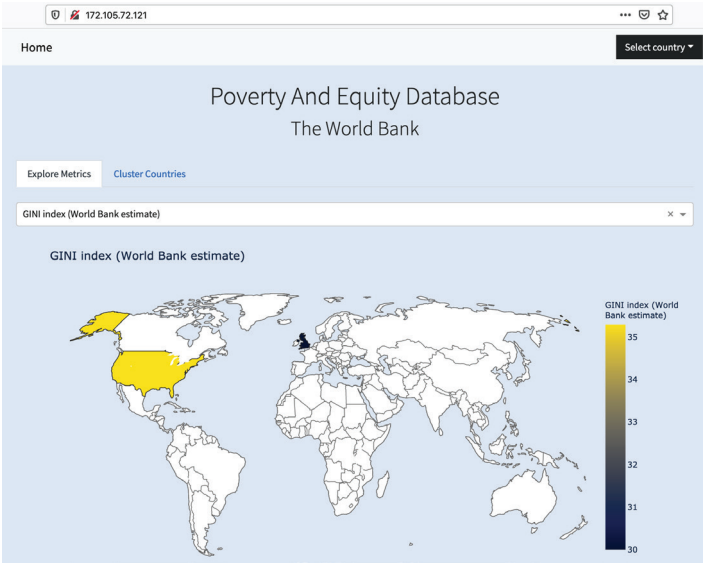


Рис. 12.9. Приложение, развернутое на публичном адресе

Поздравляем! Теперь ваше приложение доступно каждому в сети. Оно располагается на общедоступном сервере, и каждый может свободно им пользоваться.

Что обычно происходит дальше? Как вам вносить изменения в приложение и что, если вам понадобится использовать обновленные версии пакетов?

Давайте посмотрим, как можно осуществлять поддержку приложения после его развертывания.

Поддержка приложения и его обновление

После публикации приложения вам понадобится осуществлять его поддержку и обновление, и именно об этом мы поговорим в этом разделе.

Исправление ошибок и внесение изменений в приложение

Принципы при изменении приложения должны быть такими же, как и при его написании. Все внесенные изменения должны быть сначала протестирова-

ны локально, после чего переноситься в репозиторий Git. Затем мы подтягиваем изменения на сервер и перезапускаем приложение.

Обновление пакетов Python

Наше приложение использует сразу несколько дополнительных пакетов, а в рабочих проектах их количество может исчисляться десятками. Каждый пакет живет своей жизнью и может обновляться. Вам необходимо следить за тем, чтобы в вашем приложении использовались актуальные версии пакетов. В некоторых обновлениях производятся изменения, касающиеся безопасности, и такие обновления необходимо устанавливать как можно быстрее. Другие реализуют определенные изменения функционала. Для обновления пакета достаточно выполнить команду `pip install --upgrade <package_name>`, при этом вам нужно всегда проверять, не навредила ли вашему приложению обновленная версия. Обновления хорошо поддерживаемых пакетов обычно сопровождаются необходимыми инструкциями и комментариями.

Решившись на обновление пакета, вы можете выполнить его локально, чтобы убедиться, что приложение работает корректно.

1. Перейдите в командной строке в папку приложения, как показано ниже:

```
cd /path/to/your/app
```

2. Активируйте виртуальное окружение следующим образом:

```
source /bin/activate
```

3. Имя активного окружения должно появиться слева в командной строке в круглых скобках. Теперь можно обновить требующийся пакет:

```
pip install --upgrade  <package_name>
```

4. Если все прошло гладко, можете запустить свое приложение локально с помощью следующей команды и убедиться, что все работает корректно:

```
pip install --upgrade  <package_name>
```

5. Если все в порядке, нужно обновить файл `requirements.txt`, чтобы отразить новую версию пакета и другие зависимости. Сначала воспользуйтесь командой `pip freeze`, которая собирает всю информацию о версиях пакетов в текущем виртуальном окружении и выводит ее в поток стандартного вывода (`stdout`). Теперь необходимо перенаправить этот вывод в файл `requirements.txt`, чтобы перезаписалась прежняя информация. Можно выполнить эту команду в один заход, но лучше сначала ознакомиться с текущими версиями пакетов:

```
pip freeze > requirements.txt
```

6. Сохраните изменения в репозитории Git и отправьте их в GitHub. Команда `git add` позволяет добавить файл(ы) в область подготовки, где находится информация, готовая к сохранению в репозитории. После

этого нужно выполнить сохранение изменений при помощи команды `git commit`, которая также принимает текстовый комментарий с указанием произведенных действий. Команда `git push` позволяет финально сохранить изменения в онлайн-репозитории:

```
git add requirements.txt
git commit -m 'Update requirements file'
git push
```

7. Теперь, когда мы подготовили обновленный файл `requirements.txt` в центральном репозитории, можно перенести его на сервер, как мы уже делали это раньше. После подключения к серверу, перехода в папку вашего проекта и активации виртуального окружения воспользуйтесь командой `git pull`, которая позволяет делать две вещи. Сначала она просматривает последние изменения репозитория на удаленном сервере, а затем объединяет изменения в локальной копии, в результате чего вы получаете готовое приложение. Команда представлена ниже:

```
git pull
```

8. В данном случае мы работали с файлом `requirements.txt`. Теперь можно запустить процесс обновления пакетов на сервере в соответствии с версиями в этом файле:

```
pip install -r requirements.txt
gunicorn app:server
```

В результате ваше приложение со всеми внесенными изменениями будет перезапущено. Мы продемонстрировали этот процесс на примере изменения файла `requirements.txt`, но с таким же успехом мы могли внести изменения в любой исходный файл проекта или файл с данными. Это общий подход, который можно использовать для любых изменений.

Теперь, когда мы обзавелись новым компонентом инфраструктуры для нашего приложения – сервером, вам придется осуществлять и его поддержку.

Поддержка сервера

Ниже мы коротко и без подробных объяснений перечислим действия, которые вам придется время от времени выполнять применительно к вашему серверу, хранящему приложение. Для выполнения этих мероприятий вам может потребоваться изучить базовые принципы и процедуры администрирования систем на базе Linux, что вы можете сделать самостоятельно:

- **добавление пользовательского домена**: возможно, вам захочется, чтобы пользователи могли обращаться к вашему приложению по звучному и запоминающемуся имени, а не по сухому IP-адресу. Для этого вам придется приобрести доменное имя, зарегистрировать его и выполнить все необходимые мероприятия по его активации. В документации Linode есть исчерпывающие инструкции по этому поводу;

- **настройка сертификата безопасности**: это очень важная процедура, которую можно выполнить бесплатно, следуя многочисленным инструкциям;
- **обновление пакетов**: мы сделали это при первом входе на сервер, но вам необходимо периодически проделывать эти операции, чтобы поддерживать инфраструктуру приложения в актуальном состоянии, в том числе и в отношении безопасности;
- **дополнительная безопасность**: принятые нами меры безопасности ограничились созданием нового пользователя и включением его в группу sudo. Но можно пойти дальше и обезопасить доступ по SSH при помощи пар ключей аутентификации, настройки брандмауэра и других мер.

Теперь, когда вы узнали, как можно выполнить развертывание приложения Dash на базе сервера Linux, посмотрим, как можно сделать это с использованием Dash Enterprise. Мы подчеркнем преимущества этой корпоративной платформы и подробно расскажем о ее применении в главе 13.

Развертывание и масштабирование приложений Dash с помощью Dash Enterprise

В этом разделе мы коротко коснемся основных процедур развертывания приложения с помощью Dash Enterprise.

Инициализация приложения

Выполнив все необходимые подготовительные мероприятия и настройку, вы можете перейти на страницу **Application Manager** для инициализации приложения. Это можно сделать, нажав на кнопку **Initialize App** и введя имя приложения. Выбранное имя будет определять ссылку, по которой вы сможете обращаться к приложению, как показано на рис. 12.10.

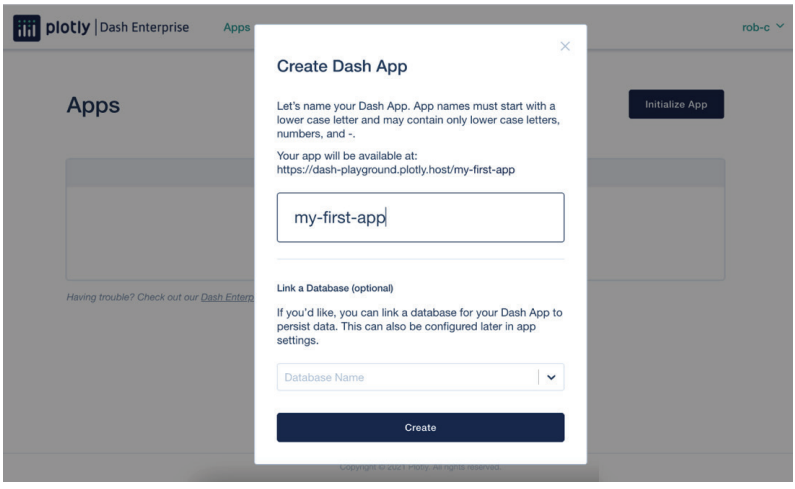


Рис. 12.10. Инициализация приложения в Dash Enterprise

Создание приложения (необязательная фаза)

Если ваше приложение уже завершено, вы можете пропустить эту фазу. В противном случае инструмент **Dash Enterprise Development Workspaces** поможет вам в построении приложения.

Кроме того, этот инструмент содержит специальное окружение, позволяющее произвести все стадии подготовки приложения от его написания и до развертывания, не покидая окна браузера. Также вам будет доступно более 80 шаблонов, которые помогут вам определиться со структурой и внешним видом вашего приложения. На рис. 12.11 показан пример созданного приложения.

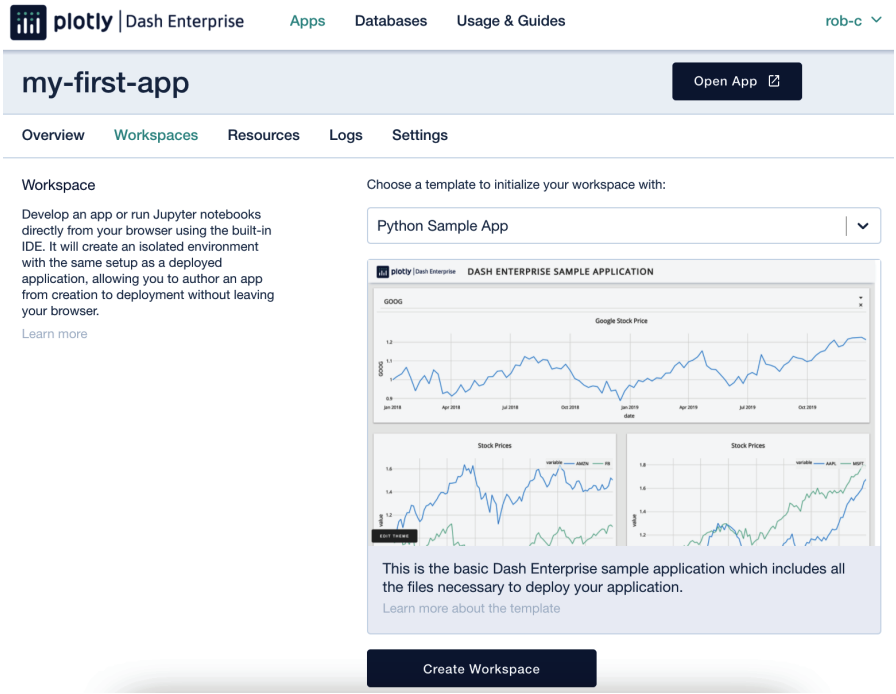


Рис. 12.11. Выбор из доступных шаблонов в Dash Enterprise

Помимо этого, в состав инструмента Workspaces входит подобная **VS Code** среда разработки с возможностью запускать Jupyter Notebooks онлайн, что видно на рис. 12.12.

Подготовка папки проекта

Папка проекта имеет схожую структуру с нашей рабочей папкой с приложением, так что здесь вам многое будет знакомо. Эта папка должна содержать три основных файла:

- `app.py` (или `index.py`) – файл с исходным кодом приложения. Файл также должен содержать переменную `server`, которая инициализируется следующим образом: `server = app.server`;

- `requirements.txt` – этот файл необходим для установки правильных версий зависимостей по примеру того, как мы делали это ранее;
- `Procfile` – это очень простой файл, содержащий команды, необходимые для запуска приложения. К примеру, в нашей схеме развертывания приложения мы использовали команду `gunicorn app:server`. В этом случае в файл `Procfile` будет добавлена строка `web: gunicorn app:server`. Сюда также могут включаться и другие команды, требующиеся для запуска приложения.

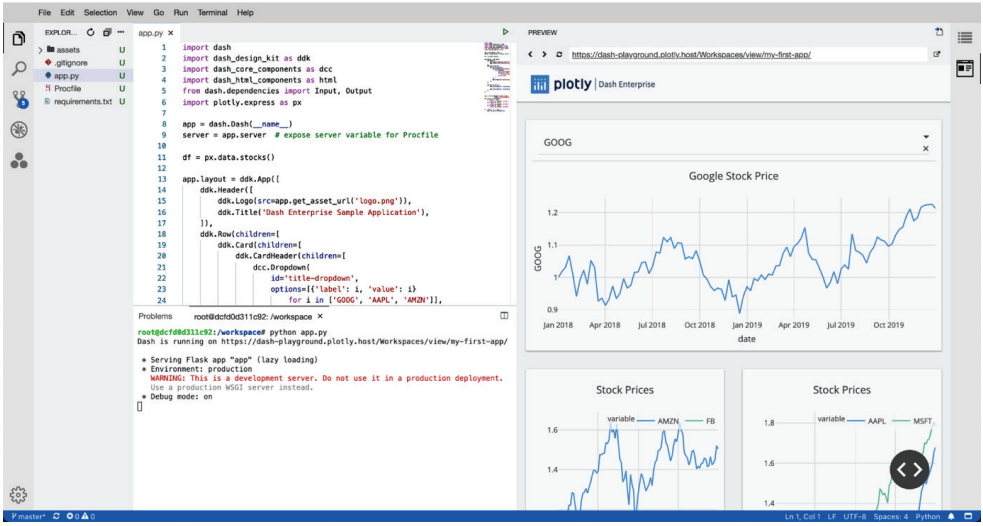


Рис. 12.12. Среда разработки в рамках Dash Enterprise

В папке проекта также могут находиться дополнительные файлы:

- `runtime.txt`: в этом файле задается требующаяся вам версия Python;
- `apt-packages`: понадобится, если ваше приложение требует установки дополнительных пакетов на уровне системы, например драйверов базы данных;
- `app.json`: необходим, если вы хотите запускать скрипты при изменении процедуры развертывания;
- `CHECKS`: потребуется, если вам необходимо настроить проверку работоспособности приложения перед выпуском.

Итак, мы инициализировали приложение, создали его и убедились, что в папке проекта находятся все нужные нам файлы. Теперь обратимся непосредственно к процедуре развертывания приложения.

Развертывание приложения в Dash Enterprise

Развертывание приложения при помощи Dash Enterprise происходит на основе Git, так что многое из перечисленного вам будет знакомо. Находясь в домашней папке приложения, выполните следующие действия.

1. Инициализируйте репозиторий Git для начала отслеживания изменений в папке приложения:

```
git init
```

2. Укажите удаленный сервер для вашего приложения. Это делается один раз, чтобы Git знал, куда посылать и откуда получать изменения:

```
git remote add plotly https://<your-dash-enterprise-domain>/
GIT/<app-name>
```

3. Добавьте файлы или папки, по которым хотите выполнить коммит:

```
git add file1 file2 ...
#или для добавления всех изменений в вашей рабочей папке:
git add .
```

4. Подтвердите изменения для выгрузки в удаленный репозиторий с кратким комментарием произведенных изменений:

```
git commit -m "Initial commit"
```

5. Выгрузите изменения в удаленный репозиторий:

```
git push plotly master
```

Операция развертывания может занять около пяти минут, после чего приложение будет доступно по указанному вами имени домена.

Как мы уже упоминали, Dash Enterprise уместно использовать для крупномасштабных развертываний, где изменения должны быть четко задокументированы и протестированы, а приложение должно пройти определенную проверку. Вспомогательные инструменты, входящие в состав Dash Enterprise, помогают облегчить этот процесс.

Каждый раз, когда вы вносите изменения в приложение, необходимо выполнить набор тестов, перед тем как двигаться дальше. Инструменты *непрерывной интеграции* (continuous integration) и *непрерывного развертывания* (continuous deployment) также являются частью Dash Enterprise, и многие из них содержат графический интерфейс, позволяющий упростить задачи. Кроме того, платформа Dash Enterprise поддерживает большинство систем аутентификации, среди которых LDAP, SAML, простая аутентификация по e-mail и другие, что облегчает процесс интеграции с информационной инфраструктурой.

В этой главе мы сделали большой шаг вперед и рассказали о множестве техник и подходов, связанных с развертыванием приложений. Давайте вспомним, о чем мы поговорили.

Заключение

Начали главу мы с описания рабочего процесса по управлению циклом разработки, развертывания и обновления приложения. Мы выделили три главных компонента в этом процессе и определились с тем, как они соотносятся друг с

другом. Также мы обсудили взаимодействие между локальной системой, центральным репозиторием Git и веб-сервером.

После этого мы создали рабочий аккаунт на хостинге Linode и подготовили его к работе. Далее наладили связь с сервером по SSH и выполнили несколько базовых задач, связанных с администрированием и безопасностью. Затем клонировали репозиторий и убедились, что приложение работает на сервере подобно локальной установке.

В дальнейшем обсудили ключевые компоненты, позволяющие сделать наше приложение доступным в сети интернет. Мы затронули такие важные темы, как обновление пакетов Python, изменение файлов, выгрузка их в репозиторий Git и отправка на сервер. Этим вам придется заниматься довольно часто.

Завершив с процессом развертывания приложения на сервере Linux, мы поговорили о высокоуровневой платформе под названием Dash Enterprise, обсудив входящие в ее состав инструменты и рассмотрели процесс развертывания приложения.

В заключительной главе книги мы обсудим направления, которые могли бы вам быть интересны, но о которых мы не рассказали в предыдущих главах. На данный момент вы уже довольно хорошо представляете себе принципы работы приложений Dash и можете самостоятельно изучить нюансы, которые мы не осветили. А мы обратим ваше внимание на некоторые интересные аспекты, с которыми вам будет полезно ознакомиться.

Глава 13

Следующие шаги

Добро пожаловать в заключительную главу книги и в начало вашего путешествия по миру Dash! В предыдущих главах мы обсудили множество тем, рассмотрели процесс создания различных визуализаций и создание интерактивных дашбордов, но Dash этим всем не ограничивается, предел в нем – это ваша фантазия.

К настоящему моменту вы должны уверенно чувствовать себя при создании полноценных дашбордов и свободно выражать данные при помощи подходящих типов диаграмм.

Но книга, которую вы держите в руках, – это лишь первый шаг на длинном пути, и в этой главе мы вкратце рассмотрим идеи и темы, которые вам необходимо будет изучить самостоятельно. Что-то из этого мы упоминали в книге, а что-то нет.

Темы, которые будут рассмотрены в главе:

- развитие навыков в области анализа и подготовки данных;
- исследование новых техник визуализации;
- знакомство с другими компонентами Dash;
- создание собственных компонентов Dash;
- реализация и визуализация моделей машинного обучения;
- повышение эффективности и использование инструментов для работы с большими данными;
- масштабирование с Dash Enterprise.

Технические требования

В этой главе мы не будем писать код или осуществлять развертывание приложений, так что никаких особых технических требований вы здесь не увидите.

Как вы знаете, существует два принципиальных подхода к овладению новой информацией и развитию навыков: *сверху вниз* (top-down approach), когда вам необходимо что-то сделать и вы ищете средства для этого, и *снизу вверх* (bottom-up approach) – в этом случае вы начинаете со знакомства с инструментами, после чего ищете варианты для их применения. Поговорим об этих подходах подробнее:

- **подход сверху вниз.** Иногда у вас возникают срочные потребности что-то реализовать, причем сделать это быстрее, лучше или легче. И для

этого вам приходится в ускоренном режиме изучать что-то новое. Ценность такого подхода заключается в его практичности. У вас есть определенные требования, и вы точно знаете, чего хотите, – это помогает вам сконцентрироваться и выбрать наиболее подходящий инструмент для решения поставленной задачи. Здесь все вращается вокруг конкретной цели. В то же время концентрация на решении задачи может не позволить вам более обширно изучить новые техники, и вы будете упускать из виду что-то важное и значимое;

- **подход снизу вверх.** Этот способ приобретения навыков направлен в обратную сторону. Здесь вы начинаете изучать что-то новое просто для общего развития, расширения профессионального кругозора или попросту из любопытства. Это может быть как обширная область вроде машинного обучения, так и узкая – например, исследование нового параметра функции, которой вы пользуетесь ежедневно. Такой подход открывает для вас новые возможности и позволяет узнать новые способы решения привычных задач, а также предусматривает заблаговременное изучение предмета – задолго до того, как его применение будет вызвано необходимостью. Знаменитая цитата «Чем больше я тренируюсь, тем чаще мне везет» подходит здесь как нельзя лучше. Преимущество этого подхода состоит в тщательности изучения предмета и лучшей теоретической подготовленности перед решением конкретных задач. Из недостатков этого способа можно отметить то, что знания в этом случае часто приобретаются в основном теоретически, в отрыве от практики, и с этим связана сложность их дальнейшего применения.

Лично я в своей практике зачастую чередую эти два подхода. Иногда я целиком сконцентрирован на решении какой-то практической задачи, и тогда ищу средства для ее решения и в процессе осваиваю их. Но временами наступает период стагнации, когда у меня иссякает фантазия и способность создавать что-то новое. Тогда я перехожу в теоретический режим. Мне доставляет истинное удовольствие изучать что-то новое для себя в такие периоды. После освоения новой для себя области знаний ко мне возвращается желание творить и воплощать приобретенные навыки на практике.

Только вам решать, какой подход лучше сработает в вашем случае. А сейчас давайте посмотрим, что еще вам будет полезно изучить, путешествуя по миру Dash.

Развитие навыков в области анализа и подготовки данных

Если вы читали введение в какую-либо книгу по *науке о данных* (data science), то наверняка знаете, что специалисты в этой области большую часть времени проводят за очисткой, переформатированием и преобразованием исходных данных.

И в этой книге мы также не раз видели это на практике. Теперь вы можете себе представить, сколько усилий и знаний в предметной области необходимо

применить просто для того, чтобы преобразовать данные в формат, пригодный для дальнейшего анализа. Но как только мы достигли своего и привели информацию в удобный вид (например, в длинный формат датафрейма), все становится намного проще.

Для приобретения полноценного набора навыков в области преобразования данных вам нужно будет в полной мере освоить пакеты `pandas` и `NumPy`. К тому же изучение новых техник, заложенных в библиотеке `pandas`, в отрыве от практического применения позволит вам развить воображение. В свою очередь, освоение регулярных выражений поможет лучше понять принципы анализа текста, поскольку обычно текст бывает неструктурированным, а извлечение и нахождение в нем определенных шаблонов делает более понятным процесс очистки данных. Навыки статистического и математического анализа также играют важную роль. В конце концов, мы здесь просто вертим цифрами!

Приобретение навыков и знаний в области манипуляции с данными неизбежно ведет к лучшему пониманию принципов их визуализации.

Исследование новых техник визуализации

Читая книгу, вы должны были заметить, каким мощным инструментом является Plotly Express и как легко с его помощью строить визуализации. Кроме того, вы обратили внимание, сколько дополнительных опций он предлагает. В то же время мы ограничены тем, что данные для эффективного представления должны быть отформатированы в нужном виде, с чем Plotly Express, увы, не поможет. И именно здесь должны пригодиться ваши навыки специалиста по анализу данных.

Мы рассмотрели четыре основных типа диаграмм, но это лишь вершина айсберга, состоящего из бесчисленного количества всех возможных видов визуализации. Здесь можно применить те же два подхода, о которых мы говорили в начале главы. Вы можете идти от конкретного требования построить определенный график и изучать именно этот тип диаграммы, а можете в целом осваивать принципы визуализации, в результате чего у вас появятся идеи относительно нового представления своих данных.

К изучению диаграмм можно подходить с разных сторон. Можно исследовать их по типам геометрических фигур и атрибутов, в виде которых они представлены, например начать с круговых и точечных диаграмм. Также вы можете разделить их по типу применения: допустим, есть статистические и финансовые диаграммы. Многие виды визуализаций сводятся к простым геометрическим формам, таким как точки, окружности, прямоугольники, линии и т. д. И отличаются они по внешнему виду и способу их комбинирования.

Также важной техникой в отношении визуализации данных является использование подграфиков. На протяжении книги мы активно применяли фасетирование, позволяющее размножить диаграммы для множественных наборов данных. Подграфики, с другой стороны, используются для создания массивов независимых диаграмм. С применением этой техники можно строить богатые отчеты с использованием одной диаграммы, где подграфики будут служить для анализа различных аспектов данных.

Изучив весь спектр техник и видов визуализации данных, вы можете добавлять их в свои приложения и делать их интерактивными.

Знакомство с другими компонентами Dash

В этой книге мы познакомились только с основными компонентами из состава Dash, но этот богатый фреймворк ими не ограничивается. В отношении получения новых знаний о составляющих Dash можно выделить три подхода:

- **получение углубленных знаний об известных вам компонентах**: мы рассмотрели достаточно много компонентов на протяжении книги, но не погружались в них до конца, а ограничились лишь их базовыми характеристиками. Вы можете восполнить этот пробел самостоятельно. К примеру, компонент `dash_table` хранит в себе много тайн и секретов, которые вам будет очень полезно постигнуть;
- **исследование новых компонентов**: как мы уже говорили, существует множество компонентов фреймворка Dash, о которых в этой книге мы не упоминали. Два из них, не требующих дополнительных описаний, – это `DatePickerSingle` и `DatePickerRange`. Компонент `Interval` позволяет выполнить код по истечении заданного временного промежутка, а `Store` – сохранить некоторые данные в браузере пользователя для ускорения/улучшения работы приложения. Если вам нужно осуществлять загрузку файлов из приложения, поможет компонент `Upload`. Все эти компоненты идут в составе пакета Dash Core Components. Существуют и другие пакеты, включающие в себя полезные компоненты. Например, библиотека `Dash Cytoscape` позволяет строить потрясающие интерактивные графы, которые мы уже несколько раз видели, когда использовали визуальный отладчик для лучшего понимания происходящего в приложении. Такие графы могут быть применимы в приложениях из самых разных областей. Чтобы позволить пользователям самим рисовать на графиках, вы можете обратиться к опциям аннотаций в изображениях в Dash или использовать инструменты из пакета `Dash Canvas`. Вместе эти средства помогут создать довольно мощные визуализации, в которых ваши пользователи буквально получат возможность рисовать с помощью мыши в свободном стиле или с применением наборов геометрических фигур;
- **изучение компонентов от сообщества**: поскольку фреймворк Dash является проектом с открытым исходным кодом и располагает инструментами для создания собственных компонентов, многие разработчики воспользовались этой возможностью, воплотив самые разные свои идеи. Одним из таких пользовательских пакетов является Dash Bootstrap Components, которым мы не раз пользовались в этой книге. Но есть и многие другие, а их список постоянно пополняется.

Это приводит нас к следующем теме…

Создание собственных компонентов Dash

Возможно, вам будет любопытно узнать, что Dash изначально проектировался как «React для Python, R и Julia». Как вы, наверное, знаете, React представляет собой очень объемный фреймворк JavaScript для построения пользовательских

интерфейсов. Есть большая библиотека компонентов React с открытым исходным кодом, и в пакете Dash Core Components, по сути, содержатся компоненты React, доступные в Python. Получается, если вам нужен какой-то особый функционал, который не реализован в Dash, вы можете создать его самостоятельно, нанять для этого сторонних разработчиков или даже спонсировать разработку такого компонента от Plotly. Некоторые из компонентов, которыми мы пользовались в этой книге, были созданы именно таким образом – путем спонсирования со стороны заказчиков. Это, кстати, один из способов поддержать группу разработчиков Dash. И это идет на пользу всем, кто использует фреймворк Dash с открытым исходным кодом.

Существуют абсолютно четкие инструкции по поводу создания своих собственных компонентов Dash, и вам наверняка будет интересно с ними ознакомиться, если вы собираетесь реализовываться в этой области. Это позволит вам лучше понять, как работает библиотека в целом, и, быть может, вы в конечном счете разработаете функционал, которым будут пользоваться многие.

С навыками манипулирования данными, их визуализации и создания новых компонентов вы можете пойти гораздо дальше элементарного построения графиков. А внедрение в свои проекты приемов машинного обучения позволит вам вывести свои модели на новый уровень и повысить интерес к ним со стороны других пользователей и разработчиков.

Реализация и визуализация моделей машинного обучения

Машинное и глубокое обучение – это, конечно, совершенно отдельные темы, но со всеми навыками, перечисленными ранее, вы сможете вывести свое понимание принципов машинного обучения на новый уровень. В конце концов, вы используете графическое представление для выражения определенных идей о ваших данных, и с хорошим багажом интерактивных визуализаций вы сможете дать своим пользователям богатые возможности для тестирования различных моделей и настройки гиперпараметров.

Повышение эффективности и использование инструментов для работы с большими данными

Это очень важная тема, поскольку мы всегда должны заботиться о том, чтобы наши приложения работали с высокой степенью эффективности. В этой книге мы не останавливались на вопросах оптимизации приложений, главным образом сосредоточившись на создании приложений Dash и их функционале. Кроме того, мы в основном работали с небольшими наборами данных размером не больше нескольких мегабайт. Но даже при взаимодействии с такими датасетами может быть важно уделять внимание эффективности приложений. Большие данные – это не обязательно большие файлы, они могут быть и маленькими, но с многократной обработкой.

Ниже мы перечислим основные способы оптимизации приложений, при этом большие данные, как мы уже сказали, являются отдельной темой, стоящей специального рассмотрения.

При наличии полного понимания поведения приложения и ресурсов, которые будут использованы в процессе его работы, вам необходимо уделить время очистке кода и данных, чтобы в них не было ничего, что могло бы негативно сказаться на эффективности приложения. Следующие моменты вы можете поправить прямо сейчас:

- **загружайте только нужные данные**: мы загружали файл целиком, а для каждого обратного вызова ограничивали датафрейм. Это может быть не лучшим вариантом. Если у нас есть обратный вызов, к примеру для данных о численности населения, мы можем создать для этого отдельный файл, а затем и отдельный датафрейм, который будет содержать только нужные столбцы, и работать именно с ним;
- **оптимизируйте типы данных**: иногда нам приходится работать с данными, содержащими многократные повторения одних и тех же значений. Например, в наборе данных с информацией об уровнях бедности названия стран повторяются много раз. С целью оптимизации хранения таких данных можно использовать категориальный тип данных в `pandas`.

1. Импортируйте модуль `sys` и обратите внимание на разницу в байтах при хранении строковых значений (названий стран)...

```
import sys
sys.getsizeof('Central African Republic')
73
```

2. ...и целочисленных:

```
sys.getsizeof(150)
28
```

3. Как видите, разница в объеме занимаемой памяти получилась почти трехкратная. Именно на такой результат можно рассчитывать при использовании категориального типа данных. В этом случае создается словарь с сопоставлениями всех строковых значений целочисленным аналогам, после чего эти значения используются для кодирования текста. В результате получается большая экономия места.

4. Загрузите набор данных `poverty`:

```
import pandas as pd
poverty = pd.DataFrame('data/poverty.csv')
```

5. Получите поднабор данных, содержащий столбец с названиями стран, и проверьте занимаемое им место в памяти:

```
poverty[['Country Name']].info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8287 entries, 0 to 8286
Data columns (total 1 columns):
    #   Column        Non-Null Count  Dtype
    ---  ------        --------------  -----
    0   Country Name  8287 non-null   object
dtypes: object(1)
memory usage: 64.9+ KB
```

6. Сконвертируйте столбец в категориальный тип данных и снова выполните проверку:

```
poverty['Country Name'].astype('category').to_frame().info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8287 entries, 0 to 8286
Data columns (total 1 columns):
    #   Column        Non-Null Count  Dtype
    ---  ------        --------------  -----
    0   Country Name  8287 non-null   category
dtypes: category(1)
memory usage: 21.8 KB
```

7. В результате простейшего преобразования типа данных столбца нам удалось снизить объем занимаемой памяти с 64,9 Кб до 21,8 Кб – примерно в три раза.

8. Также стоит задуматься об изучении и применении на практике технологий и принципов работы с большими данными. Один из главных проектов в этой области на данный момент – это `Apache Arrow`, в котором принимают совместное участие лидеры из сообществ баз данных и науки о данных. Одна из главных инициатив проекта состоит в объединении усилий сообществ из разных дисциплин и языков программирования.

9. К примеру, вам необходимо прочитать файл CSV и представить его в памяти в виде некого датафрейма. Используете ли вы язык R, Python или любой другой, ваши действия будут очень похожи и содержать операции сортировки, выбора столбцов, их фильтрации и т. д. Таким образом, будет приложено много усилий на выполнение практически одинаковых действий на разных платформах. С точки зрения производительности известно, что большая часть процессорного времени уходит на преобразование объектов из одного формата в другой, а также на операции чтения и записи. Также время тратится на запись объектов на диск и чтение их из других языков. Это приводит к истощению ресурсов и зачастую к необходимости выбирать один язык программирования, чтобы исключить межъязыковые противоречия.

10. Одной из целей Apache Arrow является создание единой структуры данных в памяти, представляющей некое подобие датафрейма. С ее помощью объекты могут передаваться между различными платформами без необходимости выполнять какие бы то ни было преобразования. Вы представляете, насколько проще может быть организовано взаимодействие в таких условиях между разными языками, особенно если в них используется одна спецификация объектов.
11. Для каждого языка программирования может быть написана соответствующая библиотека. К примеру, в Python такая библиотека именуется `pyarrow`, и вам может быть полезно ее исследовать. Она может быть использована как обособленно, так и в интеграции с `pandas`.
12. Также стоит обратить внимание на формат файлов `parquet`, входящий в состав проекта. Подобно CSV и JSON, `parquet` не зависит от языка, и файлы в этом формате могут быть прочитаны в любой платформе при наличии соответствующего инструмента. К счастью, `pandas` напрямую поддерживает работу с файлами `parquet`.
13. Одной из важнейших особенностей формата `parquet` является его способность к сжатию. Таким образом, этот формат можно считать одним из наиболее выгодных с точки зрения занимаемого места. Кроме того, файлы `parquet` обеспечивают очень эффективное чтение и запись. В файлах содержатся все необходимые метаданные и схема. Также данные в них хранятся в отдельных структурах, что повышает эффективность их чтения. Ниже перечислены основные особенности формата `parquet`:

    - **колоночное хранилище**: в отличие от ориентированных на строки файлов CSV, в формате `parquet` данные в основном хранятся в виде столбцов. Файлы с построчным хранением больше подходят для транзакционных задач. К примеру, когда пользователь заходит на сайт, нам нужно извлечь информацию о нем (строку), а затем изменить ее на основе выполненных им действий. Но в аналитической обработке если нам нужно, допустим, получить средний доход по странам, для этого достаточно считать только две колонки из набора данных. При колоночном хранении мы можем легко и быстро перемещаться в рамках столбца и гораздо более эффективно извлекать нужные нам данные по сравнению с построчным хранением. Другие столбцы мы при этом можем извлекать по мере необходимости – если нам потребуется проанализировать данные в них;
    - **кодирование**: как и в примере с категориальным типом данных в `pandas`, который мы рассмотрели ранее, формат `parquet` также предусматривает кодирование информации на основе словаря. Есть и другие системы кодирования, которые используются в этом формате. К примеру, существует кодирование на основе расхождений, или *дельта-кодирование* (delta encoding), применимое при использовании больших значений. В этом случае система хранит первое значение в колонке, а для всех последующих использует не абсолютные значения, а их разницу от первого. Допустим, список значе-

ний [1000000, 1000001, 1000002, 1000003] может быть представлен следующим образом: [1000000, 1, 1, 1]. Как видите, в исходном виде мы храним только первое значение, а для других используем приращение. Этот прием позволяет сэкономить немало места в памяти и может быть эффективно использован в столбцах с датами и временем, где в числовом представлении значения получаются большие, а разница между ними не столь существенная. При чтении данных из такого столбца программа автоматически выполняет все вычисления и возвращает вам массив информации в исходном виде. Помимо этих способов кодирования, в файлах parquet могут быть использованы и другие;

- **секционирование**: также очень интересной особенностью формата parquet является то, что данные могут храниться в виде нескольких файлов и объединяться из них при желании. Представьте себе файл с данными о людях, в котором присутствует 10 млн строк. При этом в одном из столбцов хранятся данные о половой принадлежности человека: "male" или "female". Теперь, если разбить эти данные физически на два файла, пропадет всякая необходимость хранить информацию о поле, а достаточно будет вынести ее в название файла. Таким образом, если нам понадобится обратиться к этой колонке, программа будет знать, как восстановить ее значения, поскольку все они в этом случае будут одинаковыми;
- **хранение статистики**: еще одной особенностью формата parquet является возможность хранить статистическую информацию для каждой группы в столбцах. Представьте, что у вас есть набор данных, состоящий из 10 млн строк, и вам необходимо извлечь из них те, которые входят в диапазон от 10 до 20. Предположим, все строки разбиты на группы по 1 млн строк. В заголовке каждой группы в этом случае будет храниться информация о минимальном и максимальном значениях в группе. При сканировании заголовков вам не понадобится много времени, чтобы понять, что в конкретной группе с максимальным значением 6 не могут находиться нужные вам числа. Таким образом, вы просто пропускаете 1 млн строк, не анализируя их содержимое, и для принятия этого решения вам понадобилось провести единственное сравнение.

Вы уже обрели необходимые навыки для создания сложных дашбордов с интерактивными диаграммами с использованием фреймворка Dash, однако это не добавило вам опыта развертывания действительно крупномасштабных проектов. В этом вам поможет уже знакомый вам Dash Enterprise.

Масштабирование с Dash Enterprise

При выполнении развертывания приложения в рамках крупной организации со множеством пользователей, каждый из которых располагает собственной инфраструктурой, у вас могут возникнуть разного рода сложности. Представь-

те, что вашим приложением должны ежедневно пользоваться сотни человек. Как обеспечить им доступ к приложению? Как управлять их паролями и что делать в случае увольнения или принятия в штат нового сотрудника? Есть ли у вас достаточный опыт в области безопасности данных, чтобы взять на себя ответственность за развертывание столь масштабного приложения?

В подобных случаях на первый план выходит *инженерия данных* (data engineering), заключающаяся в эффективном и безопасном хранении данных. Масштабирование и управление такими данными может стать весьма сложной задачей, если это не входит в ваши непосредственные компетенции. В конце концов, в ваши обязанности может входить создание и настройка приложений для поиска аналитических инсайтов, а не поддержка масштабных инфраструктур. Также вы можете обладать необходимыми знаниями, но не иметь желания заниматься администрированием, а склоняться к созданию интерфейсов, моделей и визуализаций.

Здесь вам на помощь придет уже знакомый по предыдущим главам Dash Enterprise. По сути, это обычный Dash, но в связке с инструментами, предназначенными для крупномасштабных развертываний.

Давайте рассмотрим некоторые полезные инструменты, входящие в состав Dash Enterprise.

Dash Design Kit

В **Dash Design Kit** включается набор инструментов для выбора дизайна и темы приложения и входящих в его состав компонентов и графиков. Этим набором можно пользоваться в условиях, когда вы вынуждены часто вносить изменения в приложение и строить многочисленные дашборды. Вероятно, вам захочется уделить больше внимания моделям и визуализации, а не выбору и установке тем. Кроме того, в крупных организациях могут присутствовать строгие требования касательно брендирования отчетов, и с помощью DDK такие задачи решаются крайне легко.

App Manager

Это основной инструмент для управления циклом развертывания приложений. Чем более объемными и частыми являются ваши релизы приложения, тем больше смысла есть в использовании **App Manager**, помогающего облегчить процесс развертывания.

Инструмент App Manager позволяет безопасно развертывать, управлять и делиться приложениями в рамках масштабируемой платформы Kubernetes.

Snapshot Engine

Это еще один инструмент, позволяющий управлять своим приложением на разных его стадиях. Как ясно из названия, **Snapshot Engine** поддерживает возможность создания ссылки на моментальный снимок приложения Dash. Вы также можете программно сохранять информацию в файлах PDF и/или отправлять отчеты по электронной почте. Кроме того, Snapshot Engine позволяет запрашивать такие снимки вручную или по расписанию.

Еще одной возможностью, предоставляемой этим инструментом, является рисование на экране при помощи мыши. Это позволяет легко создавать аннотации для коллег, не слишком грамотных технически, и делиться с ними комментариями в отчетах.

Повышение производительности с помощью Job Queue

Если в вашем приложении содержится большое количество задач для запуска и/или функций обратного вызова, требующих большого процессорного времени, имеет смысл отдельно управлять этими процедурами для повышения производительности приложения. **Job Queue** представляет собой набор инструментов, позволяющих задать расписание для входящих веб-запросов и снизить вероятность возникновения проблем в случае появления запросов в очереди на исполнение.

Корпоративная безопасность

Процесс установки Dash Enterprise является достаточно гибким и может быть произведен как в окружении *виртуального частного облака* (virtual private cloud, VPC), так и на отдельный сервер. Кроме того, как мы уже говорили в предыдущей главе, Dash Enterprise может быть интегрирован с большинством основных протоколов аутентификации, что позволяет бесшовно внедрить приложения Dash в корпоративное окружение.

Консультационная служба

Наконец, у вас есть возможность связаться с командой разработчиков Dash, обладающих богатым опытом работы с самыми разными компаниями. Они всегда открыты к возможности создания персонализированного решения, кроме того, у них за плечами действительно большой опыт, и они наверняка сталкивались с ситуацией, похожей на вашу.

Как видите, прибегать к использованию Dash Enterprise стоит в случаях, когда вам предстоит осуществлять действительно масштабное развертывание приложения, при котором вы сможете извлечь выгоду из всех входящих в эту платформу инструментов. Но вы можете все сделать самостоятельно, оценив перед этим все возможные сложности и компромиссы.

Итак, мы высказали лишь несколько идей, которыми вы можете воспользоваться, но в конечном счете все решает ваш творческий подход, знания предметной области и желание работать. Давайте подведем итоги этой главы и книги в целом.

Заключение

Начали мы эту главу с акцента на важности приобретения навыков манипуляции с данными. Эти навыки включают в себя умение преобразовывать исходные данные, очищать их и приводить в формат, пригодный для дальнейшего анализа и визуализации. Мы также упомянули различные техники визуализации и типы диаграмм, которые помогут вам наиболее эффективно выразить ваши собственные идеи.

Далее мы перечислили некоторые компоненты Dash, которым не уделили время в книге, а также отметили компоненты, которые каждый день рождает на свет сообщество. Вы и сами можете принять участие в создании новых компонентов, а я с удовольствием выполню после этого команду `pip install dash-your-components` и посмотрю, что у вас получилось!

Также мы рассмотрели применение на практике моделей машинного обучения, их визуализацию и интерактивность. Навыки манипуляции с данными помогут вам сделать ваши модели легко интерпретируемыми и полезными с точки зрения пользователей, особенно не обладающих достаточными техническими знаниями.

Кроме того, в заключительной главе мы поговорили о техниках, применимых при работе с большими данными, и обсудили особенности корпоративного развертывания приложений с помощью Dash Enterprise.

Спасибо за то, что прочитали эту книгу. Я надеюсь, она пришлась вам по душе. А я буду с нетерпением ждать ваших приложений, в которых вы воплотите свои блестящие идеи.

Предметный указатель

Книги издательства «ДМК ПРЕСС» можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
Тел.: **+7(499) 782-38-89**. Электронная почта: **books@alians-kniga.ru**.

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.galaktika-dmk.com.

Элиас Даббас

Интерактивные дашборды и приложения с Plotly и Dash

Используем полноценный веб-фреймворк
в Python на всю мощь – без JavaScript

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Гинько А. Ю.*
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 24,86. Тираж 200 экз.

Веб-сайт издательства: **www.dmkpress.com**

Фреймворк Dash предоставляет разработчикам Python блестящие возможности создания полноценных многофункциональных интерактивных веб-приложений и дашбордов без знания языка JavaScript.
Прочитав эту книгу, вы научитесь использовать потенциал визуализации Dash по максимуму. Познакомитесь с экосистемой Dash, ее основными пакетами и сторонними библиотеками. Узнаете, как создавать базовое приложение Dash и наполнять его нужным вам функционалом. Сможете интегрировать в приложение выпадающие списки, слайдеры и многое другое и связывать их с графиками и другими элементами вывода.
В итоге вы освоите навыки, необходимые для создания и развертывания полноценных интерактивных приложений и дашбордов, внесения в них нужных изменений с помощью рефакторинга кода и дополнения любым требуемым функционалом.

- Как создавать и запускать интерактивные веб-приложения и дашборды
- Как конвертировать визуализации в различные форматы, включая изображения и файлы HTML
- Как использовать модуль Plotly Express и концепцию, называемую грамматикой графиков, для сопоставления данных и визуальных атрибутов
- Как создавать различные типы диаграмм, такие как диаграмму рассеяния, линейную, столбчатую, гистограмму, карту и другие
- Как улучшить ваши приложения за счет создания динамических страниц с созданием содержимого на основе ссылок
- Как реализовывать обратные вызовы для управления графиками на основе ссылок и наоборот

Издание предназначено специалистам по работе с данными и аналитикам, желающим больше узнать о своих исходных данных при помощи интерактивных дашбордов.

Предполагается знание на базовом уровне языка программирования Python.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

Packt›

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-988-0

9 785970 609880 >