O'REILLY

# Python Polars

## Подробное руководство

### Преобразование, визуализация и анализ данных с помощью эффективных датафреймов

Йерун Янссенс и Тейс Ньюдорп

Вступительное слово от создателя Polars Ричи Винка

BOOKS.KZ

Йерун Янссенс и Тейс Ньюдорп

# Python Polars: подробное руководство

Jeroen Janssens and Thijs Nieuwdorp

# Python Polars: The Definitive Guide

## Transforming, Analyzing, and Visualizing Data with a Fast and Expressive DataFrame API

O'REILLY®

Йерун Янссенс и Тейс Ньюдорп

# Python Polars: подробное руководство

## Преобразование, анализ и визуализация данных с помощью быстрых и эффективных датафреймов

BOOKS.KZ

**УДК 004.6:004.823Polars**
**ББК 16.35**
**Я65**

**Янссенс Й., Ньюдорп Т.**
Я65 Python Polars: подробное руководство / пер. с англ. А. Ю. Гинько. – Астана: Books.kz, 2025. – 502 с.: ил.

**ISBN 978-6-01140-650-5**

В этом подробном руководстве продемонстрированы все ключевые возможности потрясающе эффективной библиотеки Polars для обработки и анализа данных на языке Python. Разнообразные примеры помогут вам быстро освоить API и перейти к практическому применению Polars в задачах, связанных с обработкой и исследованием данных, построением конвейеров и многим другим.

Книга будет полезна широкому кругу специалистов по работе с данными независимо от уровня их квалификации.

УДК 004.6:004.823Polars
ББК 16.35



ISBN 978-1-098-15608-4 (англ.)

ISBN 978-6-01140-650-5 (казах.)

# Содержание

# 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, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу *dmkpress@gmail.com*. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

## *Нарушение авторских прав*

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится 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.

# Отзывы о книге

В этой книге Йерун и Тейс проделали блестящую работу, не только рассказав читателям обо всех нюансах Polars, но также поведав о том, как легче расстаться с привычками, приобретенными при использовании других библиотек, таких как pandas. Им действительно удалось раскрыть весь потенциал выражений, лежащих в основе эффективного использования Polars и более декларативного и функционального подхода к обработке данных. Уверен, читая эту книгу, вы откроете для себя все глубины Polars и сможете взглянуть на процесс анализа данных по-новому.

*— Ричи Винк (Ritchie Vink), создатель Polars*
*(выдержка из вступительного слова)*

Библиотека Polars предстала в виде восходящей звезды на горизонте экосистемы обработки и анализа данных в Python и открыла новые границы возможностей для датафреймов следующего поколения. Йерун и Тейс очень вовремя представили общественности свою книгу, позволяющую постичь все нюансы и преимущества Polars.

*— Уэс Маккини (Wes McKinney),*
*создатель pandas и главный архитектор Posit PBC*

Библиотека Polars принесла великое множество долгожданных инноваций в мир обработки данных при помощи датафреймов в виде хорошо организованного интерфейса и эффективной реализации. Это позволило вывести процесс обработки и анализа данных в Python на новые, ранее недостижимые высоты. Мы также очень рады тому, что Ричи и его команда являются частью амстердамской экосистемы анализа данных. Меня крайне впечатляют способности Йеруна простым и понятным языком преподносить сложный материал из области науки о данных, будь то работа в командной строке или где бы то ни было еще. Их с Тейсом книга стала ярким подтверждением этого тезиса, и я горячо рекомендую ее к прочтению всем в сообществе анализа данных.

*— Ханнес Мюхлеисен (Hannes Mühleisen),*
*партнер по созданию DuckDB*

Работая совместно с Йеруном и Тейсом над миграцией конвейеров данных в Polars, мы изначально относились к этому проекту с долей скепсиса, но очень быстро убедились в быстродействии и интуитивности этой библиотеки и ее API. И пока авторы допоздна трудились над окончанием своей книги, мы делали все, чтобы улучшить наши рабочие процессы. Мы надеемся, что эта

книга поможет вам в освоении всех тонкостей работы с библиотекой Polars – такой прекрасной и такой ленивой!

*— Марникс ван Лисхаут (Marnix van Lieshout)*
*и Брам Тиммерс (Bram Timmers),*
*специалисты по работе с данными в компании Alliander*

Эта книга способна изменить ваш подход к анализу данных. Йерун и Тейс проделали феноменальную работу, тщательно выполнив все сравнения и включив необходимые диаграммы и примеры. Polars обладает очень богатым функционалом, и авторы очень бережно все разложили по полочкам. Мне импонирует их стремление визуализировать все анализируемые в книге данные с помощью привлекательных таблиц и графиков!

*— Майкл Чоу (Michael Chow),*
*главный разработчик в компании Posit PBC,*
*специалист по внедрению и поддержке Great Tables*

Эта книга очень элегантно приоткрывает покров таинственности, коим славится мощная экосистема Polars. В ней Тейс и Йерун в предельно понятной манере объясняют все теоретические основы библиотеки и сопровождают объяснения полезными примерами, что позволяет погрузиться в сложные концепции, не делая материал неподъемным. Работали ли вы ранее с pandas или только начинаете свой путь в анализе данных в Python, эта книга даст вам все необходимое для создания и осознания рабочих процессов, которые могут выполняться на предельно высокой скорости.

*— Хелла Хаанстра (Hella Haanstra),*
*инженер машинного обучения в компании Xomnia*

С библиотекой Polars я познакомился довольно давно и даже написал когда-то статью о применении метода `pipe()` к датафреймам. Меня впечатлила скорость работы этой библиотеки, но еще больше я был поражен ее API. Мне показался такой изящный подход очень прогрессивным. С тех пор прошло несколько лет, и Polars превратилась в полноценный инструмент анализа данных. Очевидно, что начинающим разработчикам понадобилось всеобъемлющее пособие по этой библиотеке. И именно его вы сейчас и держите в руках! И пособие, и справочник!

*— Винсент Д. Вармердам (Vincent D. Warmerdam),*
*специалист по работе с данными, сооснователь проекта Calmcode*

В книге *Python Polars. Подробное руководство* понятным языком рассказывается обо всех особенностях и нюансах работы с Polars. Вместе с тем она содержит огромное множество примеров практического применения этой библиотеки. Поистине отличный источник полезной информации!

*— Стин де Гоэйер (Stijn de Gooijer),*
*ведущий разработчик Polars*

Polars успел стать одним из ведущих фреймворков для работы с данными в Python, особенно в сфере анализа временных рядов и прогнозирования. В настоящее время он полностью интегрирован с такими библиотеками, как MLForecast и StatsForecast от Nixtla, позволяющими эффективно и гибко прогнозировать данные.

Йерун и Тейс проделали потрясающую работу, написав книгу, которая в равной степени подойдет как новичкам в мире анализа данных, так и профессионалам, использующим pandas, но желающим развиваться и двигаться дальше.

*— Рами Криспин (Rami Krispin),*
*старший менеджер отдела науки о данных и инженерии*
*в компании Apple*

Меня впечатлил уровень погружения в материал для написания такой феноменальной книги, проявленный Йеруном и Тейсом. Я видел несколько книг по Polars, но эта, без сомнений, лучшая! В ней авторы не просто повторяют текст документации, а подробно описывают используемые выражения и в дружеской манере сравнивают эту библиотеку с другими инструментами. Хороший пример такого подхода – описание того, как библиотека Polars из коробки работает с плагином геокодирования. Знакомы вы с Polars или нет, я в любом случае рекомендую вам прочитать эту книгу!

*— Марко Горелли (Marco Gorelli),*
*старший инженер-программист в компании Quansight,*
*ведущий разработчик Polars и pandas, создатель пакета narwhals*

# О переводчике


**Александр Гинько**, обладающий богатым опытом работы в сфере ИТ и более 15 лет посвятивший переводам книг и статей на самые разные темы, в последние годы специализируется на переводе книг в области бизнес-аналитики и программирования для издательства «ДМК Пресс» по направлениям Python, SQL, машинное и глубокое обучение, статистика, Power BI, DAX, Excel, Power Query, Tableau, R… На данный момент в активе Александра уже более 25 книг, включая одну авторскую, и он продолжает плодотворно работать над переводом и написанием новых книг.

Если вы читаете эту книгу, возможно, вам также будут интересны книги «Сверхбыстрый Python» (*https://dmkpress.com/catalog/computer/programming/python/978-5-93700-226-6*), «Python: практическое руководство по Pandas (200 упражнений)» (*https://dmkpress.com/catalog/computer/programming/python/978-5-93700-227-3*), «Введение в статистическое обучение с примерами на Python» (*https://dmkpress.com/catalog/computer/statistics/978-5-93700-217-4*) и «Инженерия данных в Python: основы анализа данных с помощью Pandas, NumPy и Scikit-Learn» (*https://dmkpress.com/catalog/computer/programming/python/978-5-93700-381-2*) в переводе Александра.

Помимо перевода книг, Александр ведет свой канал в Telegram (*https://t.me/alexanderginko_books*), на котором вы можете из первых уст получить ответы на все интересующие вас вопросы об уже переведенных книгах, находящихся в работе и запланированных на будущее. Также на канале можно найти промокоды на все книги Александра для покупки книг на сайте издательства «ДМК Пресс» с большими скидками. Купить книги Александра и следить за переводом новых книг в режиме реального времени можно с помощью его бота в Telegram по адресу *https://t.me/alexanderginko_books_bot*.

# Вступительное слово от создателя Polars Ричи Винка

Это никогда не должно было оказаться правдой, но...

В декабре 2019 года я стал отцом, после чего практически сразу началась пандемия COVID-19. Признаюсь, эти события меня прилично выбили из колеи. Я пытался не потерять рассудок и хоть как-то совмещать непривычный для меня ритм жизни новорожденного ребенка с тщетными попытками удержаться за свое собственное я. В очередном приступе родительской бессонницы я решил, что мне срочно нужен какой-то новый проект. Если честно, это было только ради развлечения.

На работе, программируя на Rust, я столкнулся с задачей объединения данных из двух файлов CSV, и оказалось, что сделать это не так-то просто. Тогда я задумался, а есть ли способы облегчения этой задачи без использования SQLite. Как и принято у программистов, очень скоро я сел за написание собственного алгоритма объединения.

В то время у меня было не так много опыта программирования на Rust, и я мало что знал об оптимизации. В общем, несмотря на распирающую меня гордость от написания своего первого алгоритма объединения данных, я не мог не признать, что по эффективности он значительно уступал аналогичной операции, выполненной в pandas. Пришедшее на смену гордости разочарование и стало тем заветным семенем, из которого по прошествии времени возник Polars.

Та моя неудача поспособствовала тому, что я начал глубоко изучать движки баз данных и погружаться в методы оптимизации, заложенные в языке Rust, что вылилось в полтора года бесконечных проб и ошибок. В процессе понимания того, как работают базы данных, какие в них используются алгоритмы, как устроена работа с памятью и т. д. мои изначальные цели изменились. Мне стало недостаточно просто опередить pandas при объединении данных, я захотел написать полноценный пакет для работы с датафреймами для Rust. Это вылилось в желание создать свой собственный высокоэффективный движок запросов, который мог бы успешно соперничать с лучшими представителями семейства движков в Python.

Свои устремления я основывал на преимуществах и недостатках pandas, декларативных подходах, использующихся в SQL и PySpark, принципах функ-

ционального программирования и строгости системы типов, характерной для Rust. Поначалу я думал, что мой движок будет основан на API pandas. Но очень быстро понял, что подобные ограничения могут негативно сказаться как на моем творческом посыле сделать что-то новое, так и на эффективности итогового продукта. Избавившись от этих сдерживающих меня пут и объединив жадный и ленивый интерфейсы в один API на основе выражений, я получил зародыш того, что сегодня именуется библиотекой Polars.

15 марта 2021 года я выложил свое творение на PyPI в качестве исследовательского проекта. После этого я постепенно превратил свое детище в полноценный пакет для работы с датафреймами. Успех проекта позволил мне в результате заручиться достаточной финансовой поддержкой и основать собственную компанию под названием Polars Inc.

Сегодня Polars – это самостоятельная компания, которой я безмерно горжусь и которая является моим нескончаемым источником энергии. Наконец-то все мои идеи могут быть воплощены в жизнь (ну, ладно, не все, но я преисполнен амбиций и не теряю надежд). В глобальном смысле в Polars я хочу воплотить идею создания единого API, который будет работать повсеместно на одном и том же стеке технологий вне зависимости от ваших требований к объему обрабатываемых данных.

Вместе с моей преданной командой в Polars Inc и постоянно растущим сообществом энтузиастов мы можем решить великое множество задумок. Каждый проект предполагает свои собственные сложности, и поиск универсальных решений для сценариев разных масштабов – дело невероятно трудное, но интересное. Одна из таких сложностей связана с обработкой временных зон. Это может звучать как пустяковая задача, но на самом деле преобразование часовых поясов с сохранением консистентности данных из разных систем – дело не из легких. Еще одна техническая сложность, с которой мы в настоящее время столкнулись, связана с преобразованием API в полноценную потоковую модель. Этот процесс требует полного переосмысления того, как различные операции, например вычисление среднего значения или доступ к последней строке, должны выполняться в контексте потока в сравнении с пакетной обработкой с хранением данных в оперативной памяти. Я уверен, что мы справимся с этими трудностями.

С момента появления библиотеки Polars ее рост превысил любые мои ожидания. Для меня было большой неожиданностью увидеть, в каких сферах применяется наше творение. К примеру, это касается его использования при симуляции метода конечных элементов с участием полутора тысяч объединений. Также я видел, что Polars используют в задачах, связанных с метапрограммированием, например при генерировании сложных запросов, которые проблематично написать вручную.

В будущее Polars я смотрю с большим воодушевлением. В первую очередь мы нацеливаемся на расширение потоковых возможностей, что позволит вам комфортно работать с огромными наборами данных на своих ноутбуках. Мы также работаем над созданием распределенной облачной среды, которая поможет быстрее выполнять большие вычисления. Кроме того, в наши планы

входит поддержка типов расширений, что, в частности, даст возможность работать с геопространственными типами данных. Как видите, мы стремимся сделать библиотеку Polars универсальным средством для работы с данными, которое в равной степени подходило бы и для анализа небольших наборов, и для обработки значительных массивов, требующих распределенных вычислений. Ну и, конечно, мы не собираемся забывать об интерфейсных улучшениях, призванных, например, заранее уведомлять пользователей о том, что запрос может завершиться неудачей.

Я действительно счастлив, что вы держите в руках эту книгу, написанную Йеруном и Тейсом. Я знаю этих парней еще по работе в Xomnia, где мы часто посещали одни и те же тренинги, выезжали вместе с компанией, отдыхали и делились идеями. Когда Йерун впервые предложил мне написать книгу, посвященную Polars, я сердечно его поблагодарил, но так и не нашел на это времени. И я очень рад, что они с Тейсом сами взялись за этот непростой труд. Как-то во время корпоративной поездки в Иорданию они буквально насели на меня, и мне пришлось на протяжении нескольких часов, сидя в автобусе, объяснять им, а иногда и с боем отстаивать, идеи, реализованные в Polars.

В результате Йерун и Тейс проделали просто фантастическую работу, не только рассказав читателям обо всех нюансах Polars, но также поведав о том, как легче расстаться с привычками, приобретенными при использовании других библиотек, таких как pandas. Им действительно удалось раскрыть весь потенциал выражений, лежащих в основе эффективного использования Polars и более декларативного и функционального подхода к обработке данных. Уверен, читая эту книгу, вы откроете для себя все глубины Polars и сможете взглянуть на процесс анализа данных по-новому.

Читайте книгу с удовольствием, самостоятельно проверяйте примеры и выводите свои навыки анализа данных на новый качественный уровень.

*— Ричи Винк (Ritchie Vink),*
*создатель Polars, Амстердам, ноябрь 2024 г.*

# Введение

Библиотека Polars заявила о себе в экосистеме обработки данных в Python как одна из самых многообещающих инноваций. Потрясающее быстродействие, выразительный API и способность с легкостью справляться с большими массивами данных очень быстро привели к тому, что о Polars заговорили как о достойном сопернике традиционных инструментов обработки данных во главе с pandas. При разработке библиотеки Polars во главу угла изначально ставилось быстродействие, чему способствует использование оптимизированных алгоритмов Rust, особенно когда речь идет о вычислениях с задействованием нескольких ядер процессора или видеокарты. Работаете ли вы с большими наборами данных, выполняете сложные преобразования или анализируете данные в реальном времени, Polars поможет вам легко справиться с самыми ресурсоемкими задачами.

Впервые мы услышали о Polars от ее создателя и нашего коллеги по компании Xomnia в Амстердаме Ричи Винка. Это было в 2022 году, когда эта библиотека еще не была так распространена, как сейчас. Послушав увлеченные рассказы Ричи об использованных в Polars технологиях, мы решили дать этой новинке шанс. И она нас не разочаровала! Мы практически сразу отметили простоту синтаксиса и впечатляющее быстродействие. Это шло вразрез с ограничениями и недостатками инструментов, которые мы в то время использовали в своих проектах. Polars стала для нас глотком свежего воздуха и поразила своей простотой и потенциалом. В общем, мы сразу отметили для себя, что это не просто очередной пакет.

Вскоре нам представилась возможность испытать Polars на практике. Мы работали над одним проектом в Alliander, крупнейшей энергоснабжающей компании в Нидерландах. В рамках этого проекта нам нужно было масштабировать один конвейер данных так, чтобы он обрабатывал все данные по всей электросети на еженедельной основе. Поскольку изначально решение было реализовано на R и Python с использованием пакета pandas, с такой нагрузкой оно справлялось с трудом. Мы провели небольшой тест с Polars и убедили команду переписать решение с использованием этой библиотеки. В результате нам удалось существенно повысить быстродействие конвейера и снизить потребление памяти. Это помогло нам не только успешно завершить проект, но и вселило безмерную веру в Polars.

Проанализировав свой успех, мы решили поделиться знаниями об этой удивительной новой библиотеке с широкой аудиторией посредством написания книги. С тех пор прошло полтора года, и вот эта книга перед вами! Мы очень надеемся, что, прочитав ее, вы сможете добиться аналогичных успехов в ваших собственных проектах.

# Для кого эта книга

Книга, которую вы держите в руках, предназначена для всех, кто собирается использовать библиотеку Polars в Python с целью более эффективного преобразования, анализа и визуализации данных. Опытный ли вы аналитик или инженер данных, либо только делаете свои первые шаги в науке о данных, в этой книге вы найдете всю необходимую информацию о том, как реализовывать на практике высокоэффективные решения, связанные с преобразованием и анализом данных. Для демонстрации того, что библиотека Polars может быть эффективно использована в самых разных сферах деятельности, представим себе двух персонажей: опытного аналитика данных Ханну и видавшего виды инженера данных Коса.

## *Ханна: аналитик данных*

Как мы уже сказали, Ханна обладает достаточным опытом в вопросах анализа данных. Она превосходно владеет Python и всесторонне использует библиотеку pandas в своих ежедневных рабочих сценариях. В то же время Ханна испытывает определенные сложности с синтаксисом pandas и чувствует, что должен быть более элегантный способ для выполнения рутинных операций. Как и многие аналитики, она часто сталкивается с необходимостью выполнения разведочного анализа данных, включающего в себя очистку, преобразование и агрегацию больших наборов данных. В процессе Ханна то и дело ловит себя на ощущении неприязни к некоторым не самым логичным синтаксическим конструкциям, особенно когда дело доходит до расширенного анализа данных или масштабирования сценариев для работы с большими наборами.

Таким людям, как Ханна, эта книга может помочь познакомиться с более эффективной и интуитивно понятной альтернативой pandas, при использовании которой работа с большими данными не потребует внесения изменений в синтаксис. Библиотека Polars предлагает более характерный для Python способ выполнения операций, входящих в ежедневный список обязанностей Ханны. Освоив Polars, она сможет значительно упростить свои рабочие процессы, начать писать элегантный код и повысить эффективность разведочного анализа данных.

## *Кос: инженер данных*

Кос – наш опытный инженер данных, в обязанности которого входит обработка больших массивов данных и построение сложных конвейеров для выполнения различных процессов по обработке данных. Он обладает большим опытом работы в Python и использует разные технологии для обеспечения плавности обработки и перемещения данных. Часто на Коса спускаются задачи по оптимизации процессов, целью которых является снижение затрат на инфраструктуру, особенно когда речь идет о больших данных. Такие задачи подразумевают снижение временных и ресурсных затрат на выполнение

сложных преобразований без необходимости управлять распределенным вычислительным кластером.

Polars может помочь Косу в решении стоящих перед ним задач. В основе этой библиотеки лежат скорость и эффективность, особенно при работе с большими данными и интенсивными преобразованиями. Параллельная модель вычислений, на которой базируется Polars, позволит Косу обрабатывать данные быстрее, чем в pandas, а интуитивно понятный API поможет сохранить простоту кода. В результате Кос сможет гораздо эффективнее решать свои задачи, связанные с инженерией данных, и масштабировать рабочие сценарии без привлечения систем распределенных вычислений с их накладными расходами и порой сложными процедурами по настройке.

### *Более широкая аудитория*

Но эта книга подойдет не только Ханне и Косу, а всем, кто так или иначе задействован в процессе написания софта для науки о данных на Python. Работаете ли вы с небольшими наборами данных или обрабатываете десятки терабайт информации, Polars позволит вам эффективно и единообразно работать с вашими источниками. Если ваша цель состоит в повышении скорости и сохранении элегантности кода при анализе и манипулировании данными без вреда для его читаемости, эта книга подойдет вам как нельзя лучше.

У книги есть свой сайт с описанием и сопроводительными материалами: *https://polarsguide.com*, а все программные коды и наборы данных, использующиеся в книге, можно загрузить с GitHub по адресу *https://github.com/jeroenjanssens/python-polars-the-definitive-guide*.

## Соглашения, используемые в книге

На протяжении книги мы будем использовать следующие соглашения:

*Курсив*
: Так мы будем выделять новые термины, пути, имена папок и файлов.

`Моноширинный`
: Моноширинным шрифтом мы будем выделять программный код, а также указания на переменные, имена функций, переменные окружения, выражения, ключевые слова и фрагменты кода в абзацах.

**`Жирный моноширинный`**
: Так мы будем показывать команды или текст, которые вам нужно ввести самостоятельно.


Так мы будем обозначать советы и предложения.


Так мы будем помечать важные примечания.


Так мы будем выделять предупреждения.

# Благодарности

Написание этой книги было для нас увлекательным делом, которое было бы невозможно без поддержки и воодушевления со стороны многих людей.

В первую очередь мы хотели бы поблагодарить Ричи Винка, создателя библиотеки Polars, за его готовность делиться всеми своими идеями и видением технологий. Его непрекращающаяся инновационная работа над Polars стала для нас настоящим источником вдохновения, а стремление углубляться в детали в разговорах позволило и нам лучше разобраться в используемых в библиотеке концепциях. Мы очень признательны Ричи за написание вступительного слова к этой книге!

Также мы выражаем глубокую признательность нашему редактору-консультанту в O'Reilly Саре Грей (Sarah Grey). Ее блестящее знание языка, ценные советы и своевременные замечания, безусловно, пошли на пользу книге. Хочется сказать спасибо рецензенту издательства Аарону Блэку (Aaron Black) за веру в наш проект и выпускающему редактору Джонатону Оуэну (Jonathon Owen) за доведение работы до конца. Работа с издательством O'Reilly была сплошным удовольствием, и мы можем отдельно выделить Дэвида Фурато (David Futato), Эмилию Филип (Emilia Philip), Карен Монтгомери (Karen Montgomery), Кейт Дуллеа (Kate Dullea), Кристен Браун (Kristen Brown), Миа Сандвик (Miah Sandvik), Шэрон Кордесс (Sharon Cordesse), Соню Саруба (Sonia Saruba) и многих других сотрудников издательства, имена которых мы не упомянули.

Особая благодарность нашим техническим редакторам Браму Тиммерсу (Bram Timmers), Кристин Стингер (Christine Stinger), Дэвиду Лангервельду (David Langerveld), Джеймсу Мейлсу (James Males), Марко Горелли (Marco Gorelli) и Стину де Гоэйеру (Stijn de Gooijer) за тщательный анализ и конструктивную критику. Ваши опыт и внимание к деталям помогли книге стать лучше. Конечно, всю вину за возможные оставшиеся в книге ошибки и неточности мы полностью принимаем на себя.

Также хочется поблагодарить наших коллег по компании Xomnia за создание окружения, в котором было приятно учиться и работать. Именно здесь Ричи начал работать над Polars, здесь он подружился с авторами книги, и здесь же родилась идея книги. Отдельно хотим поблагодарить Шерил Зандвлиет (Cheryl Zandvliet), Хорди Хомпса (Jordi Hompes) и Тима Паува (Tim Paauw) за поддержку. Кроме того, мы благодарим всех в компании Alliander за возможность реализации их проекта на Polars в виде эксперимента, который оказался удачным.

В процессе написания книги мы активно работали с компаниями NVIDIA и Dell Technologies в области оценки быстродействия Polars на GPU. Результаты этих исследований приведены в приложении. Отдельно благодарим Кристал Кук (Crystal Cook), Дилана Филкинса (Dylan Filkins), Ирину Шеховцову (Irina Shekhovtsova), Джамиля Семаана (Jamil Semaan), Йенна Йонемицу (Jenn Yonemitsu), Марка Каи (Mark Cai), Ника Бекера (Nick Becker) и Тревиса Уэллса (Travis Wells) за возможность воплотить эти идеи в жизнь. Особая благодарность Логану Лоулеру (Logan Lawler) за обеспечение нас всеми необходимыми аппаратными средствами.

Не в последнюю очередь мы хотим выразить благодарность всему сообществу Polars. Ваши энтузиазм и обратная связь не только помогли сделать эту библиотеку лучше, но и вдохновили нас на полное погружение в используемые в ней технологии. Отдельно мы хотим выделить следующие имена людей из сообщества: Адам Джонсон (Adam Johnson), Алекс Бирч (Alex Birch), Александр Биди (Alexander Beedie), Александру Бернеа (Alexandru Bernea), Алвин Ваньеки (Alvin Wanyeki), Энди Терра (Andy Terra), Арно Веннин (Arnaud Vennin), Барбера Дрост (Barbera Droste), Брэдли Грант (Bradley Grant), Карлос Шейдеггер (Carlos Scheidegger), Кристиан Хайнц (Christian Heinze), Дин Макгрегор (Dean MacGregor), Дом Ярнелл (Dom Yarnell), Фриц ван дер Вуд (Frits Van der Woude), Герт Хульселманс (Gert Hulselmans), Гийом Ришар (Guillaume Rischard), Ханнес Мюхлеисен (Hannes Mühleisen), Хассан Кибиридж (Hassan Kibirige), Хелла Хаанстра (Hella Haanstra), Изабель Фернандес Эскапа (Isabel Fernandez Escapa), Джейк Вандерплас (Jake VanderPlas), Джеймс Пауэлл (James Powell), Йерун Сиберс (Jeroen “Junior” Siebers), Джон Сандалл (John Sandall), Йоско де Бур (Josko de Boer), Лиам Бранниган (Liam Brannigan), Маркос Детри (Marcos Detry), Марникс ван Лиесхут (Marnix van Lieshout), Марисиа Винкельс (Marysia Winkels), Мэтт Харрисон (Matt Harrison), Мэтт Шепит (Matt Shepit), Майкл Чоу (Michael Chow), Майн Четинкая-Рундел (Mine Çetinkaya-Rundel), Олли Даппер (Ollie Dapper), Орсон Петерс (Orson Peters), Оуэн Прауф (Owen Prough), Петер Ванг (Peter Wang), Рами Криспин (Rami Krispin), Ричард Янноне (Richard Iannone), Романо Вакка (Romano Vacca), Томас Аархольт (Thomas Aarholt), Томас М. Ахерн (Thomas M. Ahern), Томара Янгблад (Tomara Youngblood), Винсент Д. Вармердам (Vincent D. Warmerdam), Уэс Маккини (Wes McKinney), Уильям ван Лит (William van Lith), Ю Ри Тан (Yu Ri Tan) и Юки Какегава (Yuki Kakegawa).

Помимо этого, мы бы хотели сердечно поблагодарить членов наших семей и друзей. Йерун благодарит свою супругу Эстер, дочь Флориен и сына Оливье за терпение в связи с его занятостью днями и ночами в будни и в выходные.

Тейс выражает благодарность Пауле за поддержку, помогавшую брать в руки карандаш и продолжать работать в самые сложные моменты, за несчетное количество кружек чая, согревших его душу, и за терпение при выслушивании бесконечных хвалебных од и жалоб в адрес Polars.

Наконец, мы благодарим наших читателей за интерес к Polars. Написание этой книги стало возможно только благодаря вам. Она – ваша. Надеемся, что она позволит вам подняться на новые высоты в области анализа и обработки данных.

# Об изображении на обложке

На обложке книги изображена иберийская, или пиренейская, рысь (Lynx pardinus). Это один из четырех видов рыси. Обитает исключительно на Пиренейском полуострове на юге Европы. Иберийская рысь характеризуется желтовато-коричневым мехом с темными пятнами и светлым брюхом, а также черными кисточками на ушах. Этот вид рыси имеет средний размер, взрослые особи достигают в длину 80–100 см. Подобно американской рыси и другим видам, иберийская рысь славится коротким хвостом, достигающим не более 15 см в длину.

Изначально иберийская рысь была распространена на всей территории Испании и Португалии, но во второй половине XX века этот вид потерял около 80 % популяции по причине браконьерства, разрушения среды обитания и снижения популяции их главной добычи в живой среде – дикого кролика. К 2000 году иберийская рысь оказалась на грани вымирания – на тот момент в южной Испании оставалось всего 94 особи. Благодаря попыткам экологов удалось восстановить популяцию иберийской рыси примерно до двух тысяч особей, после чего они были возвращены в прежний ареал обитания. В результате этих усилий статус иберийской рыси изменился с вымирающего на вид, подверженный риску.

Многие животные, представленные на обложках книг издательства O'Reilly, находятся на грани вымирания, и мы выступаем за их сохранение.

Автором рисунка на обложке, выполненного на основе штриховой гравюры из книги *Natural History of Animals*, является Карен Монтгомери (Karen Montgomery). Дизайн принадлежит Эди Фридману (Edie Freedman), Элли Фолкхаузен (Ellie Volckhausen) и Карен Монтгомери. На оригинальной обложке использованы шрифты Gilroy Semibold и Guardian Sans. Шрифт оригинального текста – Adobe Minion Pro, шрифт заголовков – Adobe Myriad Condensed, шрифт кода – Ubuntu Mono от студии Dalton Maag.

ЧАСТЬ I

# Начало

# Глава 1

# Введение в Polars

В 2022 году мы работали над одним сложным проектом у наших клиентов. Их система настолько разрослась, что поток данных практически вышел из-под контроля. Реализован проект был на Python и R, при этом основной упор при манипулировании данными делался на библиотеку pandas. Со временем нам удалось выявить три основные проблемы проекта:

- код усложнился до такой степени, что поддерживать его стало просто невозможно;
- быстродействие снизилось до неприличного уровня;
- потребление памяти выросло и перевалило за 500 Гб.

Эти проблемы в совокупности поставили под угрозу жизнедеятельность системы и довели затраты на инфраструктуру до предельной отметки.

В то время библиотека Polars еще была практически неизвестна, но мы уже экспериментировали с ней и отметили очень многообещающие результаты. Убедить всех участников проекта попробовать перенести код, написанный на pandas и R, на рельсы Polars было непросто, но когда мы все же сделали это и все переписали, результат нас просто ошеломил. Новый конвейер данных оказался намного быстрее, а потребление памяти снизилось с 500 Гб до 40 Гб!

Это был настоящий успех, позволивший нам на практике убедиться в быстродействии Polars. По прошествии какого-то времени мы приступили к написанию этой книги, чтобы поделиться с вами всем, что узнали за время работы с этой потрясающей библиотекой.

В этой вводной главе вы узнаете:

- об основных компонентах Polars;
- почему Polars так эффективна и популярна;
- чем Polars отличается от других пакетов для обработки данных;
- почему вам стоит использовать Polars;
- как организована эта книга;
- почему мы сосредоточились на Python Polars.

В дополнение мы продемонстрируем возможности Polars на небольшом примере, в котором преобразуем, проанализируем и визуализируем данные, относящиеся к поездкам на прокатных велосипедах в Нью-Йорке.

# Что из себя представляет Polars

*Polars* – это высокоэффективный пакет для обработки данных, написанный с целью облегчения работы с большими объемами информации. Начавшись как домашний проект Ричи Винка для изучения языка Rust и лучшего понимания процессов обработки данных[1], сегодня Polars стал одним из самых популярных пакетов. Аналитики, инженеры данных и разработчики активно используют его для анализа и визуализации данных, а также реализации высокоэффективных приложений для работы с данными.

## Ключевые особенности

Ниже перечислены некоторые ключевые особенности библиотеки под названием Polars:

- *скорость и эффективность*: будучи написанной на Rust после десятков проведенных исследований в области использования движков баз данных, библиотека Polars просто не могла иметь иные ориентиры, кроме как быстродействие и эффективность. Благодаря параллельной обработке и применению оптимизированных техник доступа к памяти эта библиотека способна обрабатывать большие наборы данных значительно быстрее в сравнении с другими пакетами, и разница может исчисляться порядками;
- *структура датафрейма*: в основе библиотеки Polars лежат датафреймы. *Датафрейм* (*DataFrame*) представляет собой двумерную структуру данных, состоящую из строк и столбцов, как в большинстве табличных процессоров и баз данных. При этом датафреймы в Polars являются неизменяемыми, что позволяет применять к ним функциональные операции и поддерживать потоковую безопасность;
- *API на основе выражений*: в Polars используется интуитивно понятный и лаконичный синтаксис для выполнения операций по обработке данных, что облегчает его изучение, использование и поддержку.

## Основные концепции

В этой книге мы опишем следующие ключевые концепции, использующиеся в библиотеке Polars:

- *ленивые вычисления*: в Polars применяется концепция ленивых, или отложенных, вычислений, при которой сначала строится оптимальный

[1] Подробнее об этом можно почитать во вступительном слове к книге.

план выполнения запроса, а запускается он только тогда, когда это действительно необходимо. Этот подход позволяет минимизировать количество ненужной работы и приводит к значительному повышению быстродействия;

- *выражения*: в Polars для определения операций, применяемых к датафреймам, используются так называемые выражения. Эти выражения являются компонуемыми и позволяют пользователям создавать сложные конвейеры данных, построенные на основе простых строительных блоков;
- *оптимизация запросов*: в Polars производится автоматическая оптимизация планов выполнения запросов, нацеленная на минимизацию использования ресурсов, в основе которой лежат выражения и характеристики данных.

## Преимущества

Основные преимущества библиотеки Polars перечислены ниже:

- *быстродействие*: благодаря применению эффективных алгоритмов, движка параллельных вычислений и векторизации на основе принципа *SIMD* (одиночный поток команд, множественный поток данных – single instruction, multiple data) библиотека Polars способна использовать преимущества современного аппаратного обеспечения по максимуму, а при необходимости дальнейшего ускорения операций может воспользоваться вычислительными ресурсами графических процессоров NVIDIA (сравнительный анализ быстродействия с применением GPU мы приводим в приложении);
- *эффективное использование памяти*: в процессе работы библиотека Polars требует меньшего потребления ресурсов памяти в сравнении с другими пакетами для работы с данными;
- *высокая совместимость*: будучи построенной на основе колоночного формата хранения данных Apache Arrow, библиотека Polars славится повышенной совместимостью с другими инструментами и пакетами для обработки данных. Она может использоваться в Rust напрямую, но имеет и языковые привязки к Python, R, SQL, JavaScript и Julia. Совсем скоро мы объясним, почему мы в этой книге сфокусировались именно на реализации Polars в Python;
- *потоковые возможности*: Polars может обрабатывать данные блоками, что позволяет использовать внешнюю память при работе с наборами данных, не помещающимися в оперативной памяти.

Обобщая, можно сказать, что Polars представляет собой мощный пакет для преобразования и анализа данных, идеально подходящий для конвейеров, в которых ключевую роль играют быстродействие и эффективность.

# Почему вам стоит использовать Polars

«Пришел за скоростью, остался из-за API» – так часто говорят в сообществе Polars. В этой фразе отражены два основных достоинства библиотеки: быстродействие и удобство использования. Давайте быстро опишем их, а затем поговорим о популярности Polars.

## Быстродействие

В первую очередь вам необходимо дать Polars шанс по причине ее высокой эффективности. На рис. 1.1 с помощью диаграммы показана скорость (в секундах) выполнения 20 разных запросов на разных фреймворках. Диаграмму можно читать построчно и смотреть время выполнения на каждом запросе. Запросы были взяты из стандартного набора для оценки быстродействия и включают в себя операции чтения с диска.

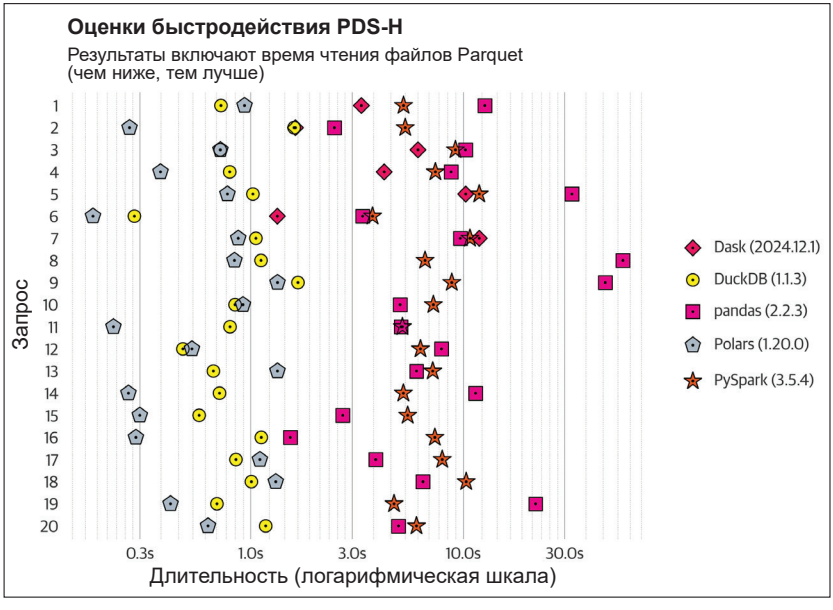



**Рис. 1.1** ❖ Сравнение быстродействия Polars с другими фреймворками[1]

Как видите, Polars чаще всего опережает по быстродействию другие фреймворки, такие как Dask, DuckDB, pandas и PySpark.

[1] Dask провалил запросы с 8-го по 20-й, а pandas в запросах 8 и 9 показал результаты 56.3 и 46.5 соответственно. Подробности можно посмотреть по адресу *https://github.com/TNieuwdorp/polars-benchmark*.

## Удобство использования

Хотя быстродействие – это действительно то, зачем к Polars обращается большинство разработчиков, многие из них в итоге остаются привержены этой библиотеке по причине ее весьма удобного и дружественного интерфейса. Основные отличительные особенности API Polars:

- *единообразие*: операции в Polars выполняются предсказуемо вне зависимости от используемых типов и структур данных и базируются на хорошо знакомом вам синтаксисе;
- *использование выражений*: в Polars реализована своя собственная система выражений, позволяющая создавать сложные схемы преобразования данных простым и удобным способом;
- *функциональный подход*: в API Polars применяются принципы функционального программирования, отлично подходящие для обработки данных и делающие ваш код простым для написания, чтения и поддержки;
- *жадный и ленивый API*: вы можете выбирать и при необходимости переключаться между жадной парадигмой выполнения операций, позволяющей быстро и ситуативно получить требуемые результаты, и ленивой парадигмой, служащей для оптимизации планов выполнения запросов. В конце этой главы мы продемонстрируем оба принципа выполнения на примерах.

Комбинация из высокого быстродействия и удобства использования позволила Polars в кратчайшие сроки собрать большую армию поклонников.

## Популярность

Никогда не стоит использовать те или иные фреймворки и библиотеки только по причине их большой популярности. Это может привести к тому, что вы пропустите инструменты, гораздо лучше подходящие вашим конкретным требованиям, поскольку популярность не является мерилом полезности и универсальности. С другой стороны, если использовать в работе малоизвестные фреймворки, вы можете столкнуться с проблемами, связанными с недостаточной поддержкой, рисками в отношении безопасности и несвоевременными обновлениями, что не позволит вам применять эти инструменты на постоянной основе на протяжении длительного времени.

К счастью, библиотека Polars очень активно поддерживается разработчиками (новые релизы на GitHub по адресу *https://github.com/pola-rs/polars/releases* появляются почти каждую неделю), а ее сообщество растет впечатляющими темпами (на сервере Discord в данный момент насчитывается более 5000 подписчиков). По опыту можно сказать, что разработчики действительно очень быстро фиксят баги и внимательно относятся ко всем предложениям.

Трудно придумать универсальное мерило популярности проектов с открытым исходным кодом. Однако количество звезд на GitHub можно считать

хорошим индикатором интереса со стороны сообщества. Оно объективно отражает количество людей, считающих пакет заслуживающим внимания и потенциально полезным. На рис. 1.2 показана динамика по количеству звезд на GitHub для разных пакетов для обработки данных в Python.

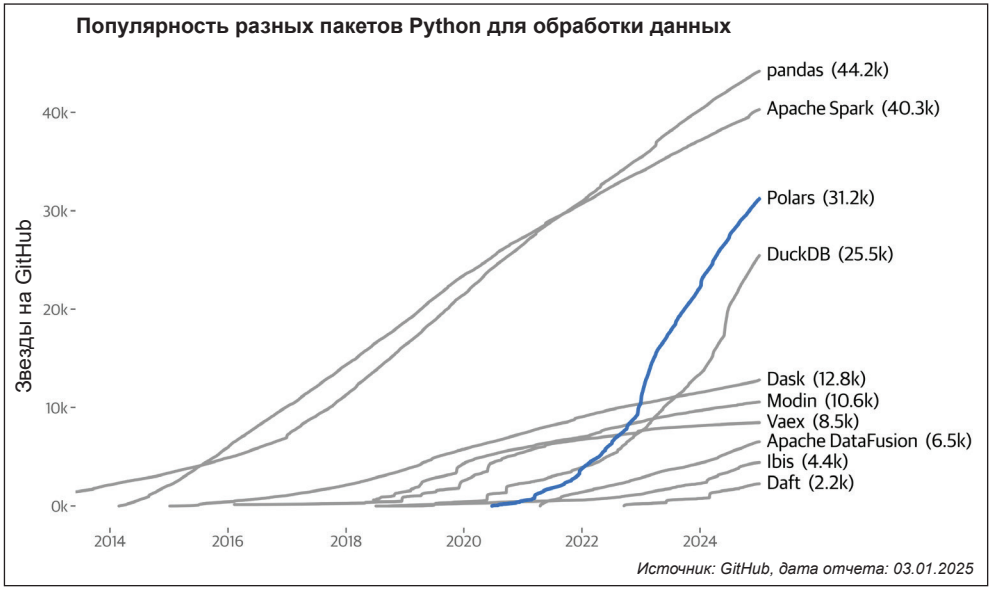



**Рис. 1.2** ❖ Популярность разных пакетов для обработки данных в Python, выраженная в количестве звезд на GitHub

Как видите, лидерами по количеству звезд являются Apache Spark и pandas – пакеты, присутствующие на рынке уже более десяти лет. В то же время третье место по популярности занимает библиотека Polars, появившаяся позже остальных. Если представить, что темпы динамики сохранятся, можно предположить, что уже в ближайшие годы Polars нагонит и опередит нынешних лидеров. В общем, Polars пришла, чтобы остаться, а остальное покажет время.

## Рациональное использование ресурсов

Принципы реализации библиотеки Polars способствуют ее высокой производительности при выполнении запросов. Согласно исследованию, проведенному Феликсом Нарстедтом в 2024 году (*https://dl.acm.org/doi/abs/10.1145/3661167.3661203*), Polars потребляет 63 % энергии, затрачиваемой пакетом pandas, при выполнении оценочных запросов теста TPC-H, и использует в восемь раз меньше энергии в сравнении с pandas на синтетических данных. При постоянном увеличении объемов обрабатываемых данных

вопрос рациональности использования ресурсов выходит на первый план. Таким образом, можно сказать, что Polars установила новые стандарты в сфере обработки данных с точки зрения пользы для экологии.

# Polars в сравнении с другими пакетами для обработки данных

Конечно, Polars – не единственная библиотека для анализа и обработки данных. В этом разделе мы представим сравнительный анализ разных пакетов, работающих в среде Python. Мы подчеркнем сильные и слабые стороны каждой библиотеки, что поможет вам понять, для каких именно задач больше подойдет Polars.

## *Pandas*

*Pandas* представляет собой наиболее популярную библиотеку для анализа и работы с данными в Python. Ее основными структурами данных являются датафреймы и объекты Series, а API содержит великое множество функций и методов для анализа, очистки и преобразования данных.

В сравнении с pandas библиотека Polars предлагает более высокую скорость обработки, особенно при работе с большими наборами данных. Эта библиотека полностью написана на языке Rust и в качестве механизма хранения данных и управления памятью использует платформу Apache Arrow, что позволяет сделать процесс анализа данных более быстрым в сравнении с pandas. Тогда как в pandas по умолчанию используются алгоритмы жадных вычислений, Polars предлагает на выбор жадную или ленивую парадигму, позволяющую выполнить оптимизацию запросов. В то же время pandas попрежнему обладает более обширной экосистемой и лучшей интеграцией с другими пакетами для обработки данных.

## *Dask*

*Dask* представляет собой фреймворк для параллельных вычислений в Python. По сути, он расширяет возможности пакетов NumPy, pandas и scikit-learn для работы в распределенных системах вычислений. Особенно хорошо Dask зарекомендовал себя при работе с большими наборами данных, не помещающимися в памяти.

Как и Dask, Polars поддерживает технологию параллельных вычислений и комфортно чувствует себя при работе с большими массивами данных. В то же время Polars предназначена для работы на одном компьютере, тогда как родной стихией Dask являются распределенные вычисления. Polars показывает высокое быстродействие применительно к операциям, умещающимся в памяти, а Dask наилучшие результаты демонстрирует в распределенной среде вычислений.

### *DuckDB*

*DuckDB* – это внутрипроцессная СУБД SQL OLAP. Она была разработана для быстрого и эффективного выполнения аналитических запросов к структурированным данным. DuckDB может быть встроена непосредственно в приложение и поддерживает запросы SQL.

И Polars, и DuckDB оптимизированы для выполнения аналитических запросов и демонстрируют на этом поприще высокое быстродействие. При этом в Polars используется API в питоновском стиле, тогда как в DuckDB для запросов применяется язык SQL.

### *PySpark*

*PySpark* представляет собой API в Python для работы с платформой Apache Spark – распределенной средой вычислений для обработки больших данных. Этот интерфейс предлагает богатый функционал, включающий в себя запросы SQL, алгоритмы машинного обучения и обработку графов. PySpark идеально подходит для обработки очень больших массивов данных в распределенной системе со множеством кластеров.

Тогда как PySpark спроектирована для распределенных вычислений, Polars реализована для работы на одной машине. Polars помогает повысить эффективность обработки данных, помещающихся в память одного компьютера, а PySpark лучше себя чувствует при работе с огромными массивами данных в распределенной среде с кластерами. Наконец, Polars легче установить и настроить в сравнении с PySpark, обладающей более обширной экосистемой.

## Почему мы остановились на реализации Polars для Python

Вы уже знаете, что библиотека Polars целиком написана на Rust и имеет языковые привязки к Python, R, SQL, JavaScript и Julia. Так почему же в этой книге мы сосредоточили свое внимание на API Polars для Python?

Согласно проведенному в 2024 году Stack Overflow исследованию среди разработчиков[1], Python является самым популярным языком программирования среди начинающих программистов и занимает четвертое по популярности место среди профессиональных разработчиков. И это неудивительно, поскольку Python славится своей простотой, читаемостью и универсальностью и широко распространен в области науки о данных, машинного обучения, веб-разработки и прочих сферах.

Такая популярность языка получила свое отражение и в сообществе приверженцев Polars. В результате API для Python является наиболее полным,

[1] С подробностями исследования можно ознакомиться по адресу *https://survey.stackoverflow.co/2024*.

популярным и часто обновляемым. Кроме того, Python повсеместно используется для анализа и обработки данных, в связи с чем большинство специалистов по работе с данными и инженеров данных с ним хорошо знакомы.

# Организация книги

В этой книге содержится 18 глав, объединенных в пять частей, и приложение. Каждая глава начинается с небольшого введения и завершается ключевыми идеями, озвученными в главе.

### *Часть I. Начало*

Первая часть книги, в которой вы сейчас и находитесь, состоит из трех глав. Эти главы призваны познакомить вас с Polars, помочь установить библиотеку и начать с ней работать.

В главе 1, которую вы сейчас читаете, мы в общих чертах рассказываем о Polars, описываем ее преимущества над другими библиотеками, а также показываем небольшой пример ее использования. В главе 2 вы получите полноценную инструкцию по установке Polars и узнаете, как получить доступ ко всем фрагментам кода и наборам данных, используемым в книге. Если у вас уже есть опыт работы в pandas, то в главе 3 вы узнаете, как проще всего осуществить плавный переход на Polars и какие привычки придется оставить в прошлом.

### *Часть II. Форматы и структуры*

Во второй части книги мы подробно поговорим о типах данных и структурах, используемых в Polars, а также обсудим техники формирования датафреймов на основе различных источников. Иными словами, вы научитесь читать и записывать данные и узнаете, как эти данные хранятся и обрабатываются в Polars.

В главе 4 мы познакомимся с используемыми в Polars структурами и типами данных, а также затронем тему обработки пропущенных значений. В главе 5 мы поговорим о разнице между жадным API, позволяющим получать результаты мгновенно, и ленивым API, используемым с целью оптимизации запросов. Глава 6 будет посвящена чтению и записи данных в различных форматах, включая CSV, Parquet и Arrow.

### *Часть III. Выражения*

Выражения играют ключевую роль в Polars, так что неудивительно, что эта часть располагается ровно посередине книги.

Глава 7 содержит примеры использования выражений, формальное определение выражений и описание способов их создания. В главе 8 мы продолжим изучать выражения и научимся выполнять математические операции, работать с пропущенными значениями и применять скользящие операции

и агрегирование. В главе 9 мы научимся комбинировать выражения с использованием, например, арифметической или булевой логики.

### *Часть IV. Преобразования*

Хорошо поняв, как работают выражения, вы сможете включать их в функции и методы с целью преобразования ваших данных. И именно этому будет посвящена четвертая часть книги.

В главе 10 мы посмотрим, как можно выбирать и создавать столбцы, а также поработаем с именами столбцов и так называемыми селекторами. Глава 11 будет посвящена различным способам фильтрации и сортировки строк. В главе 12 мы поработаем с текстовыми, временными и вложенными типами данных, а в главе 13 посмотрим, как можно группировать и агрегировать данные. Глава 14 будет посвящена разным типам объединения датафреймов, а в заключительной 15-й главе этой части мы узнаем, как можно быстро и легко менять форму представления данных путем их свертывания и развертывания различными способами.

### *Часть V. Расширенные техники*

В пятой части книги мы собрали все темы, не вошедшие в другие части.

Главу 16 мы посвятим визуализации данных при помощи разных пакетов, включая Altair, hvPlot и plotnine. В главе 17 посмотрим, как можно расширить возможности Polars путем написания собственных функций на Python и использования своих плагинов на Rust. В главе 18 мы немного приоткроем капот Polars и заглянем внутрь – посмотрим, как там все устроено и работает, и узнаем, почему это все работает так быстро.

Книга завершается приложением, в котором мы узнаем, как можно использовать ресурсы графического процессора для еще большего ускорения работы Polars.

## Пример на основе процесса ETL

Теперь, когда вы узнали, откуда появилась библиотека Polars и как она может вам помочь, пришло время посмотреть на нее в действии. Для этого мы подготовили небольшой процесс ETL, на примере которого продемонстрируем возможности Polars в области преобразования, анализа и визуализации данных, относящихся к поездкам на велосипедах в Нью-Йорке.

Строго говоря, процесс *ETL* включает в себя стадии извлечения (*E*xtract), преобразования (*T*ransform) и загрузки (*L*oad) данных, но мы добавили в наш сценарий два бонуса в виде визуализации данных.

Общий план будет такой. Сначала мы импортируем требующиеся пакеты, после чего загрузим сырые данные. Далее очистим загруженные данные

и обогатим их за счет новых столбцов. В завершение мы сохраним данные в формате Parquet, чтобы можно было повторно использовать их в следующих главах книги.

**Не беспокойтесь о синтаксисе**

Цель этого первого примера состоит в том, чтобы позволить вам почувствовать вкус Polars. Конечно, мы будем использовать множество функций и методов, которые вам совершенно незнакомы. Но не переживайте. В следующих главах книги мы подробно разберем все, что вам понадобится. Кроме того, вы не должны все приведенные здесь фрагменты кода выполнять вручную. Просто следите за сюжетом и получайте удовольствие от процесса.

Итак, приступим!

# Извлечение

Первым делом в любом процессе ETL мы *извлекаем* данные. Мы воспользуемся двумя разными источниками: в одном будут содержаться сами поездки на велосипедах, а во втором – информация об округах (borough) и районах (neighborhood)[1]. Но сначала нам необходимо импортировать все нужные пакеты, которые понадобятся для нашего примера.

## *Импорт пакетов*

Очевидно, что нам понадобится библиотека Polars. Для применения географических операций в этом примере мы воспользуемся собственным плагином, который вы можете импортировать как polars_geo. В главе 17 вы узнаете, как скомпилировать и установить этот плагин. Также нам понадобится пакет *plotnine*, которым мы воспользуемся для создания визуализаций:

```
import polars as pl ❶
import polars_geo
from plotnine import * ❷
```

❶ Обычно импортируемому пакету Polars назначается алиас `pl`.

❷ В скриптах на языке Python не рекомендуется импортировать все функции из пакетов в глобальное пространство имен, как мы здесь сделали. Но для нашего примера это допустимо, поскольку позволит использовать функции из пакета plotnine без указания имени библиотеки, что довольно удобно.

Теперь перейдем к следующему шагу с загрузкой и извлечением данных о поездках на велосипедах.

---

[1] Для простоты повествования мы будем называть borough округами, а neighborhood – районами, хотя формально это не до конца соответствует действительности. – *Прим. перев.*

## Загрузка и извлечение данных

В этом примере мы воспользуемся данными о поездках на прокатных велосипедах в Нью-Йорке с сайта компании Citi Bike. Эта система позволяет арендовать велосипеды для коротких поездок длительностью до 30 или 45 минут в зависимости от вашего уровня членства. Все эти данные находятся в свободном доступе на сайте *https://citibikenyc.com/system-data*.

С помощью показанных ниже команд в терминале мы можем загрузить архив, извлечь из него файл CSV в директорию *data/citibike/* и удалить архив за ненадобностью:

```
! curl -sO https://s3.amazonaws.com/tripdata/202403-citibike-tripdata.csv.zip
! unzip -o 202403-citibike-tripdata.csv.zip "*.csv" -x "*/*" -d data/citibike/
! rm -f 202403-citibike-tripdata.csv.zip
```

Здесь мы воспользовались командами оболочки *curl*, *unzip* и *rm*.

Эти команды оболочки не являются кодом на языке Python. В Jupyter восклицательный знак (!) перед строкой означает, что она должна быть выполнена в командной оболочке, а не в интерпретаторе Python. Если вы работаете на Windows или вам неудобно запускать команды оболочки таким образом, вы можете загрузить и распаковать архив вручную. Для этого откройте сайт *https://citibikenyc.com/system-data*, щелкните по ссылке *downloadable files of Citi Bike trip data*, загрузите файл *202403-citibike-tripdata.csv.zip*, распакуйте его и поместите полученный файл CSV в директорию *data/citibike*.

Теперь нам необходимо загрузить данные из файла CSV в датафрейм Polars.

## Загрузка данных в датафрейм Polars

Перед загрузкой сырых данных в датафрейм (DataFrame) необходимо сначала их исследовать. Подсчитаем количество строк в файле CSV с помощью команды оболочки *wc* и выведем на экран первые шесть строк командой *head*:

```
! wc -l data/citibike/202403-citibike-tripdata.csv
! head -n 6 data/citibike/202403-citibike-tripdata.csv
2663296 data/citibike/202403-citibike-tripdata.csv

"ride_id","rideable_type","started_at","ended_at","start_station_name","start_s...
"62021B31AF42943E","electric_bike","2024-03-13 15:57:41.800","2024-03-13 16:07:...
"EC7BE9D296FFD072","electric_bike","2024-03-16 10:25:46.114","2024-03-16 10:30:...
"EC85C0EEC95157BB","classic_bike","2024-03-20 19:20:49.818","2024-03-20 19:28:0...
"9DDE9AF5606B4E0F","classic_bike","2024-03-13 20:31:12.599","2024-03-13 20:40:3...
"E4446F457328C5FE","electric_bike","2024-03-16 10:50:11.535","2024-03-16 10:53:...
```

Как видите, в нашем наборе данных более 2.6 млн строк, а каждая строка представляет собой отдельную поездку на велосипеде. Похоже, что файл CSV хорошо отформатирован, в качестве разделителей используются запятые и есть заголовки.

Когда мы первый раз пытались загрузить этот файл в датафрейм Polars, то обнаружили два проблемных столбца. Значения в столбцах `start_station_id` и `end_station_id`, которые по своей сути являются текстовыми, по умолчанию загружаются в Polars как числовые, поскольку в первых нескольких строках они именно так и выглядят. Решить эту проблему можно с помощью явного указания типа данных для этих столбцов при их загрузке. Давайте прочитаем данные из нашего файла CSV в датафрейм с именем `trips` и выведем на экран количество строк в нем:

```
trips = pl.read_csv( ❶
    "data/citibike/202403-citibike-tripdata.csv",
    try_parse_dates=True,
    schema_overrides={
        "start_station_id": pl.String,
        "end_station_id": pl.String,
    },
    ).sort( ❷
        "started_at"
    )

trips.height
```

❶ Подробнее о чтении данных мы будем говорить в главе 6.
❷ О сортировке строк мы будем детально говорить в главе 11.

Вывод:

```
2663295
```

Теперь давайте взглянем на наш датафрейм. Поскольку он получился достаточно широким, чтобы уместиться на страницу книги, мы вызовем функцию *`print()`* трижды для последовательного вывода всех столбцов:

```
print(trips[:, :4])
print(trips[:, 4:8])
print(trips[:, 8:])
```

Вывод:

```
shape: (2_663_295, 4)
┌──────────────────┬──────────────┬─────────────────────┬─────────────────────┐
│ ride_id          ┆ rideable_type ┆ started_at          ┆ ended_at            │
│ ---              ┆ ---           ┆ ---                 ┆ ---                 │
│ str              ┆ str           ┆ datetime[μs]        ┆ datetime[μs]        │
╞══════════════════╪══════════════╪═════════════════════╪═════════════════════╡
│ 9EC2AD5F3F8C8B57 ┆ classic_bike  ┆ 2024-02-29 00:20:27.570 ┆ 2024-03-01 01:20:22.196 │
│ C76D82D96516BDC2 ┆ classic_bike  ┆ 2024-02-29 07:54:34.223 ┆ 2024-03-01 08:54:12.611 │
│ B4C73C958C65FEA6 ┆ electric_bike ┆ 2024-02-29 08:47:09.664 ┆ 2024-03-01 09:47:02.393 │
│ E23F7822B3D53E2A ┆ classic_bike  ┆ 2024-02-29 09:57:07.150 ┆ 2024-03-01 10:57:00.848 │
│ B0B6437C50C3AB3E ┆ electric_bike ┆ 2024-02-29 10:29:41.981 ┆ 2024-03-01 11:29:21.539 │
│ …                ┆ …             ┆ …                   ┆ …                   │
│ 197C0ABDD3348135 ┆ classic_bike  ┆ 2024-03-31 23:55:37.938 ┆ 2024-03-31 23:59:08.301 │
```

```
| 702FEBD6D9CCE4BC | classic_bike  | 2024-03-31 23:55:40.087 | 2024-03-31 23:57:26.335 |
| ECA4FC65950ADDDB | electric_bike | 2024-03-31 23:55:41.173 | 2024-03-31 23:57:25.079 |
| D8B20517A4AB7D60 | classic_bike  | 2024-03-31 23:56:17.935 | 2024-03-31 23:57:18.475 |
| 6BC5FAFEAC948FB1 | electric_bike | 2024-03-31 23:57:16.025 | 2024-03-31 23:59:22.134 |
```

shape: (2_663_295, 4)

```
| start_station_name          | start_station_id | end_station_name           | end_station_id |
| ---                         | ---              | ---                        | ---            |
| str                         | str              | str                        | str            |
| 61 St & 39 Ave              | 6307.07          | null                       | null           |
| E 54 St & 1 Ave             | 6608.09          | null                       | null           |
| FDR Drive & E 35 St         | 6230.04          | null                       | null           |
| E 6 St & Ave B              | 5584.04          | null                       | null           |
| Eastern Pkwy & Brooklyn Ave | 3871.02          | null                       | null           |
| …                           | …                | …                          | …              |
| E 59 St & Madison Ave       | 6801.01          | 3 Ave & E 62 St            | 6762.04        |
| Amsterdam Ave & W 119 St    | 7727.07          | Morningside Ave & W 123 St | 7741.01        |
| S 4 St & Wythe Ave          | 5204.05          | S 3 St & Bedford Ave       | 5235.05        |
| Division St & Bowery        | 5270.08          | Division St & Bowery       | 5270.08        |
| Montrose Ave & Bushwick Ave | 5068.02          | Humboldt St & Varet St     | 4956.02        |
```

shape: (2_663_295, 5)

```
| start_lat | start_lng  | end_lat  | end_lng   | member_casual |
| ---       | ---        | ---      | ---       | ---           |
| f64       | f64        | f64      | f64       | str           |
| 40.7471   | -73.9028   | null     | null      | member        |
| 40.756265 | -73.964179 | null     | null      | member        |
| 40.744219 | -73.971212 | null     | null      | member        |
| 40.724537 | -73.981854 | null     | null      | member        |
| 40.66939  | -73.94514  | null     | null      | member        |
| …         | …          | …        | …         | …             |
| 40.763505 | -73.971092 | 40.763126 | -73.965269 | member        |
| 40.808625 | -73.959621 | 40.81    | -73.955151 | member        |
| 40.712996 | -73.965971 | 40.712605 | -73.962644 | member        |
| 40.714193 | -73.996732 | 40.714193 | -73.996732 | member        |
| 40.707678 | -73.940297 | 40.703172 | -73.940636 | member        |
```

Для начала неплохо. Разнообразие столбцов в нашем датафрейме впечатляет – здесь есть и даты со временем, и категории, и названия, и даже координаты. Это позволит нам произвести полноценный анализ данных и построить визуализации.

## *Чтение информации о районах из GeoJSON*

Нью-Йорк – очень большой город с огромным количеством районов (neighborhood), разделенных на пять больших округов, или боро (borough): Бронкс

(The Bronx), Бруклин (Brooklyn), Манхэттен (Manhattan), Стейтен Айленд (Staten Island) и Куинс (Queens). В нашем датафрейме информация о районах и округах отсутствует. Если бы мы могли добавить в каждую строку округ и район начала и окончания поездки, это позволило бы нам сравнить разные округа на предмет частоты использования в них велосипедов или ответить на вопрос о том, в каком районе Манхэттена прокатные велосипеды пользуются наибольшей популярностью.

Для добавления этой информации мы будем читать файл в специальном формате *GeoJSON*, содержащий описание и координаты всех округов и районов Нью-Йорка. Фрагмент исходных данных, которые вы можете загрузить с GitHub по адресу *https://github.com/HodgesWardElliott/custom-nyc-neighborhoods*, выглядит так[1]:

```
! python -m json.tool data/citibike/nyc-neighborhoods.geojson
```

```
{
    "type": "FeatureCollection",
    "crs": {
        "type": "name",
        "properties": {
            "name": "urn:ogc:def:crs:OGC:1.3:CRS84"
        }
    },
    "features": [
        {
            "type": "Feature",
            "properties": {
                "neighborhood": "Allerton",
                "boroughCode": "2",
                "borough": "Bronx",
                "X.id": "http://nyc.pediacities.com/Resource/Neighborhood/Allerton"
            },
            "geometry": {
                "type": "Polygon",
                "coordinates": [
                    [
                        [
                            -73.84859700000018,
                            40.871670000000115
                        ],
                        [
                            -73.84582253683678,
                            40.870239076236174
                        ],
                        [
...
```

[1] Исходное имя файла: *custom-pedia-cities-nyc-Mar2018.geojson*, а мы переименовали его в nyc-neighborhoods.geojson.

Этот файл с глубокой вложенной структурой содержит всю нужную нам информацию. Площади здесь хранятся в виде *полигонов*, представляющих собой последовательность координат. Нам необходимо преобразовать эту вложенную структуру в табличный вид, т. е. в датафрейм:

```
neighborhoods = (
    pl.read_json("data/citibike/nyc-neighborhoods.geojson")
    .select("features")
    .explode("features") ❶
    .unnest("features")
    .unnest("properties")
    .select("neighborhood", "borough", "geometry")
    .unnest("geometry")
    .with_columns(polygon=pl.col("coordinates").list.first())
    .select("neighborhood", "borough", "polygon")
    .filter(pl.col("borough") != "Staten Island") ❷
    .sort("neighborhood")
)

print(neighborhoods)
```

❶ Об изменении формы представлений мы будем подробно говорить в главе 15.
❷ В округе Стейтен Айленд (Staten Island) отсутствуют станции проката Citi Bike.

Вывод:

```
shape: (258, 3)
┌──────────────────┬──────────┬──────────────────────────────────┐
│ neighborhood     ┆ borough  ┆ polygon                          │
│ ---              ┆ ---      ┆ ---                              │
│ str              ┆ str      ┆ list[list[f64]]                  │
╞══════════════════╪══════════╪══════════════════════════════════╡
│ Allerton         ┆ Bronx    ┆ [[-73.848597, 40.87167], [-73.…  │
│ Alley Pond Park  ┆ Queens   ┆ [[-73.743333, 40.738883], [-73…  │
│ Arverne          ┆ Queens   ┆ [[-73.789535, 40.599972], [-73…  │
│ Astoria          ┆ Queens   ┆ [[-73.901603, 40.76777], [-73.…  │
│ Bath Beach       ┆ Brooklyn ┆ [[-73.99381, 40.60195], [-73.9…  │
│ …                ┆ …        ┆ …                                │
│ Williamsburg     ┆ Brooklyn ┆ [[-73.957572, 40.725097], [-73…  │
│ Windsor Terrace  ┆ Brooklyn ┆ [[-73.980061, 40.660753], [-73…  │
│ Woodhaven        ┆ Queens   ┆ [[-73.86233, 40.695962], [-73.…  │
│ Woodlawn         ┆ Bronx    ┆ [[-73.859468, 40.900517], [-73…  │
│ Woodside         ┆ Queens   ┆ [[-73.900866, 40.757674], [-73…  │
└──────────────────┴──────────┴──────────────────────────────────┘
```

Итак, у нас есть датафрейм с 258 районами Нью-Йорка, округами, которым они принадлежат, и их полигонами, т. е. последовательностью координат их границ. Если район содержит несколько отдельных друг от друга площадей (множественные полигоны), он будет присутствовать в датафрейме несколько раз. Перед тем как добавлять информацию о районах в наш датафрейм с поездками, давайте немного поиграем в художника и визуализируем данные, чтобы ухватить необходимый контекст.

# Бонус: визуализация районов Нью-Йорка и расположения станций велосипедов

С целью визуализации районов Нью-Йорка и расположения станций, с которых арендаторы начинают свой путь, воспользуемся графическим пакетом *plotnine* (*https://plotnine.org*). Этот пакет ожидает, что на вход мы будем подавать датафрейм в длинном формате, в котором каждой координате соответствует отдельная строка, так что мы применим метод *explode()*, который позволит нам этого добиться:

```
neighborhoods_coords = (
    neighborhoods.with_row_index("id")
    .explode("polygon")
    .with_columns(
        lon=pl.col("polygon").list.first(),
        lat=pl.col("polygon").list.last(),
    )
    .drop("polygon")
)

print(neighborhoods_coords)
```

Вывод:

```
shape: (27_569, 5)
┌─────┬──────────────┬─────────┬────────────┬──────────┐
│ id  ┆ neighborhood ┆ borough ┆ lon        ┆ lat      │
│ --- ┆ ---          ┆ ---     ┆ ---        ┆ ---      │
│ u32 ┆ str          ┆ str     ┆ f64        ┆ f64      │
╞═════╪══════════════╪═════════╪════════════╪══════════╡
│ 0   ┆ Allerton     ┆ Bronx   ┆ -73.848597 ┆ 40.87167 │
│ 0   ┆ Allerton     ┆ Bronx   ┆ -73.845823 ┆ 40.870239│
│ 0   ┆ Allerton     ┆ Bronx   ┆ -73.854559 ┆ 40.859954│
│ 0   ┆ Allerton     ┆ Bronx   ┆ -73.854665 ┆ 40.859586│
│ 0   ┆ Allerton     ┆ Bronx   ┆ -73.856389 ┆ 40.857594│
│ …   ┆ …            ┆ …       ┆ …          ┆ …        │
│ 257 ┆ Woodside     ┆ Queens  ┆ -73.910618 ┆ 40.755476│
│ 257 ┆ Woodside     ┆ Queens  ┆ -73.90907  ┆ 40.757565│
│ 257 ┆ Woodside     ┆ Queens  ┆ -73.907828 ┆ 40.756999│
│ 257 ┆ Woodside     ┆ Queens  ┆ -73.90737  ┆ 40.756988│
│ 257 ┆ Woodside     ┆ Queens  ┆ -73.900866 ┆ 40.757674│
└─────┴──────────────┴─────────┴────────────┴──────────┘
```

Для получения координат станций мы для каждого названия станции отправления вычислим медианные значения широты и долготы:

```
stations = (
    trips.group_by(station=pl.col("start_station_name"))
    .agg( ❶
        lon=pl.col("start_lng").median(),
        lat=pl.col("start_lat").median(),
    )
```

```
    .sort("station")
    .drop_nulls()
)

print(stations)
```

❶ Подробности об агрегировании данных вы узнаете в главе 13.

Вывод:

```
shape: (2_143, 3)
┌─────────────────────────────┬────────────┬────────────┐
│ station                     ┆ lon        ┆ lat        │
│ ---                         ┆ ---        ┆ ---        │
│ str                         ┆ f64        ┆ f64        │
╞═════════════════════════════╪════════════╪════════════╡
│ 1 Ave & E 110 St            ┆ -73.938203 ┆ 40.792327  │
│ 1 Ave & E 16 St             ┆ -73.981656 ┆ 40.732219  │
│ 1 Ave & E 18 St             ┆ -73.980544 ┆ 40.733876  │
│ 1 Ave & E 30 St             ┆ -73.975361 ┆ 40.741457  │
│ 1 Ave & E 38 St             ┆ -73.971822 ┆ 40.746202  │
│ …                           ┆ …          ┆ …          │
│ Wyckoff Ave & Stanhope St   ┆ -73.917914 ┆ 40.703545  │
│ Wyckoff St & 3 Ave          ┆ -73.982586 ┆ 40.682755  │
│ Wythe Ave & Metropolitan Ave┆ -73.963198 ┆ 40.716887  │
│ Wythe Ave & N 13 St         ┆ -73.957099 ┆ 40.722741  │
│ Yankee Ferry Terminal       ┆ -74.016756 ┆ 40.687066  │
└─────────────────────────────┴────────────┴────────────┘
```

Осталось построить диаграмму. В следующем фрагменте кода мы воспользуемся пакетом plotnine и построим географическую диаграмму, на которой точками выделим станции велосипедов. Четырьмя цветами мы обозначим четыре округа. Цветовое заполнение районов мы используем просто для их визуального разделения, никакого иного смысла в этом не будет:

```
(
    ggplot(neighborhoods_coords, aes(x="lon", y="lat", group="id"))
    + geom_polygon(aes(alpha="neighborhood", fill="borough"), color="white")
    + geom_point(stations, size=0.1)
    + scale_x_continuous(expand=(0, 0))
    + scale_y_continuous(expand=(0, 0, 0, 0.01))
    + scale_alpha_ordinal(range=(0.3, 1))
    + scale_fill_brewer(type="qual", palette=2)
    + guides(alpha=False)
    + labs(
        title="Районы Нью-Йорка и станции Citi Bike",
        subtitle="2143 станции на 106 районов",
        caption="Источник: https://citibikenyc.com/system-data",
        fill="Округ",
    )
    + theme_void(base_family="Verdana", base_size=14)
    + theme(
        dpi=300,
        figure_size=(7, 9),
        plot_background=element_rect(fill="white", color="white"),
```

```
        plot_caption=element_text(style="italic"),
        plot_margin=0.01,
        plot_title=element_text(ha="left"),
    )
)
```

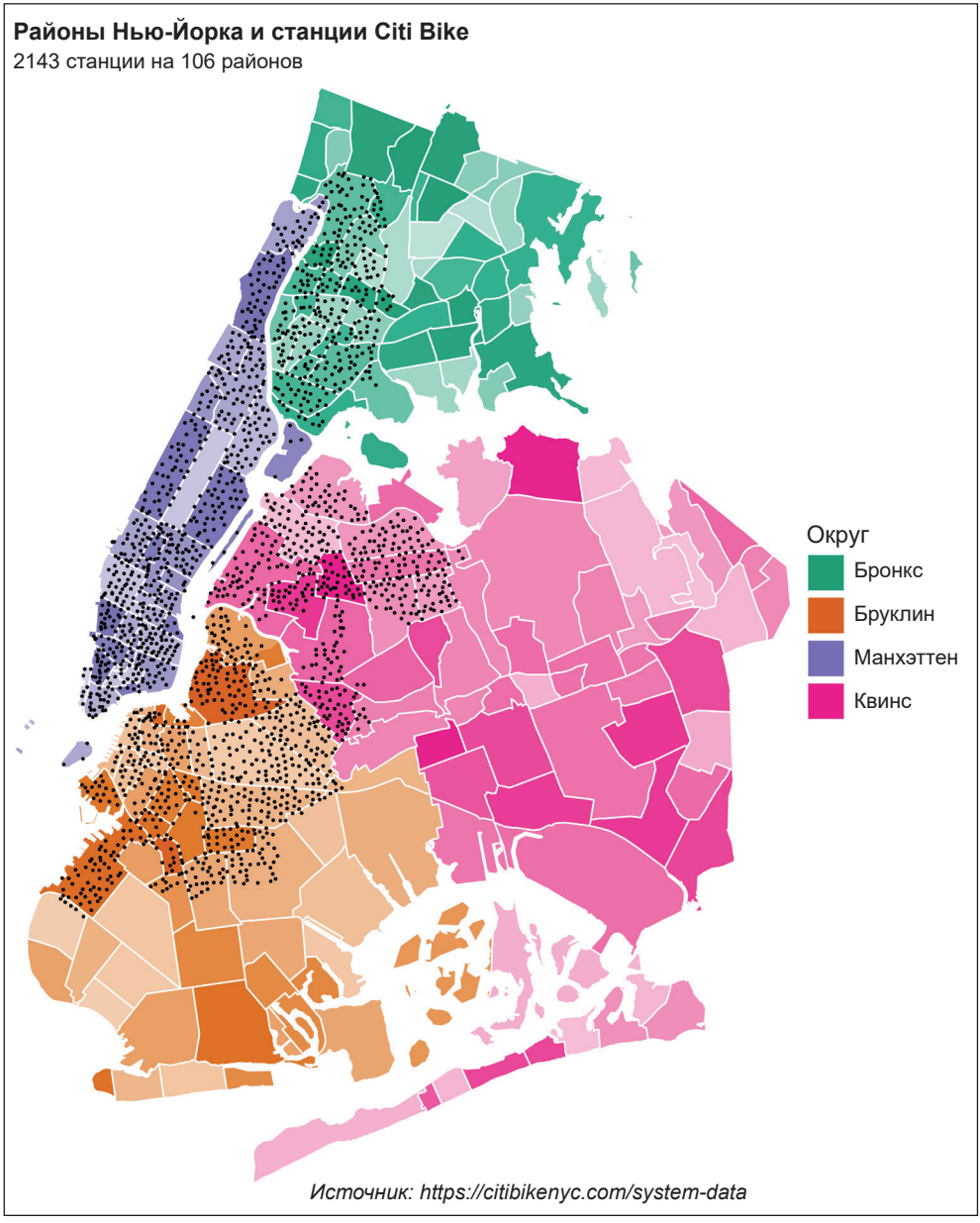



**Рис. 1.3** ❖ Районы Нью-Йорка и станции Citi Bike

Неужто Нью-Йорк не великолепен?!

# Преобразование

Ни один набор данных не идеален, и наш – не исключение. Именно поэтому в процедуре ETL второй шаг относится к преобразованию (*T*ransform) данных. Начнем со столбцов и постепенно дойдем до строк. Попутно мы добавим несколько столбцов, тем самым обогатив наш набор данных.

## *Очистка столбцов*

В приведенном ниже фрагменте кода выполняется очистка столбцов в датафрейме `trips`, под которой подразумевается следующее:

- избавляемся от столбцов `ride_id`, `start_station_id` и `end_station_id` за ненадобностью;
- сокращаем имена столбцов, чтобы с ними было легче работать;
- приводим столбцы `bike_type` и `rider_type` к категориальному типу, что лучше отражает суть хранящихся в них значений;
- добавляем столбец `duration` на основе столбцов с началом и окончанием поездки.

```
trips = trips.select(
    bike_type=pl.col("rideable_type")
    .str.split("_")
    .list.get(0)
    .cast(pl.Categorical), ❶
    rider_type=pl.col("member_casual").cast(pl.Categorical),
    datetime_start=pl.col("started_at"),
    datetime_end=pl.col("ended_at"),
    station_start=pl.col("start_station_name"),
    station_end=pl.col("end_station_name"),
    lon_start=pl.col("start_lng"),
    lat_start=pl.col("start_lat"),
    lon_end=pl.col("end_lng"),
    lat_end=pl.col("end_lat"),
).with_columns( ❷
    duration=(pl.col("datetime_end") - pl.col("datetime_start"))
)

trips.columns
```

❶ О приведении типов вы подробнее узнаете в главе 7.
❷ О выборе и создании столбцов мы будем детально говорить в главе 10.

Вывод:

```
['bike_type',
 'rider_type',
 'datetime_start',
 'datetime_end',
 'station_start',
```

```
 'station_end',
 'lon_start',
 'lat_start',
 'lon_end',
 'lat_end',
 'duration']
```

Что мы здесь сделали, если кратко? В столбце `rideable_type` разбили текст по символу подчеркивания, взяли только первый элемент (`classic`, `electric`, ...) и сохранили в столбце с именем `bike_type`. Привели столбец `member_casual` к категориальному типу и сохранили в виде столбца с именем `rider_type`. Переименовали еще несколько столбцов, а также создали столбец с именем `duration` на основе разницы между значениями в столбцах `datetime_end` и `datetime_start`.

Теперь перейдем к очистке строк.

## *Очистка строк*

Вы могли заметить, что в нашем наборе данных присутствуют пропущенные значения. Поскольку у нас достаточно много данных, мы можем относительно безболезненно избавиться от пропусков. Если данных у вас не так много, вам придется использовать более изощренные стратегии с заменой пропусков средними или наиболее часто встречающимися значениями.

Есть у нас в наборе данных и поездки, начавшиеся в феврале, а завершившиеся в марте. Для анализа и визуализации будет лучше, если мы также избавимся от таких записей. Наконец, мы удалим поездки, которые начинались и заканчивались на одной станции и продолжались менее пяти минут, т. к. нас не интересуют такие странные передвижения:

```
trips = (
    trips.drop_nulls()
    .filter( ❶
        (pl.col("datetime_start") >= pl.date(2024, 3, 1))
        & (pl.col("datetime_end") < pl.date(2024, 4, 1))
    )
    .filter(
        ~(
            (pl.col("station_start") == pl.col("station_end"))
            & (pl.col("duration").dt.total_seconds() < 5 * 60)
        )
    )
)

trips.height
```

❶ О фильтрации строк вы подробнее узнаете в главе 11.

Вывод:

```
2639170
```

После этих преобразований в нашем наборе данных `trips` осталось более 2.6 млн строк, что вполне достаточно.

## *Добавление информации о дистанции поездки*

Нам было бы интересно знать, какую дистанцию преодолевают арендаторы велосипедов, поскольку в дальнейшем мы могли бы сравнить эти данные со временем поездки. Конечно, у нас нет в наличии подробных данных о траекториях поездок всех велосипедистов, и максимум, что мы можем сделать, – это взять начальную и конечную точки поездки и вычислить по ним расстояние с помощью функции гаверсинуса (haversine distance)[1].

Функцию гаверсинуса можно было бы реализовать средствами Polars, но мы бы хотели воспользоваться так называемым контейнером, или *крейтом* (crate), с именем geo. Здесь есть один момент: дело в том, что этот крейт создан в Rust, а не в Python. Так что мы создали отдельный пакет специально для этой книги для преобразования крейта geo в плагин для Polars. Это позволило нам вычислять расстояние между объектами на сфере так, как если бы мы пользовались родным методом Polars.

Метод `Expr.geo.haversine_distance()` принимает на вход координату в виде широты и долготы:

```
trips = trips.with_columns(
    distance=pl.concat_list("lon_start", "lat_start").geo.haversine_distance(
        pl.concat_list("lon_end", "lat_end")
    )
    / 1000 ❶
)

trips.select(
    "lon_start",
    "lon_end",
    "lat_start",
    "lat_end",
    "distance",
    "duration",
)
```

❶ Результат расчета расстояния с помощью функции гаверсинуса исчисляется в метрах, так что мы делим его на 1000, чтобы перевести в километры. Подробнее о пользовательских плагинах мы поговорим в главе 17.

Вывод:

```
shape: (2_639_170, 6)
┌────────────┬──────────┬───────────┬──────────┬──────────┬─────────────────┐
│ lon_start  ┆ lon_end  ┆ lat_start ┆ lat_end  ┆ distance ┆ duration        │
│ ---        ┆ ---      ┆ ---       ┆ ---      ┆ ---      ┆ ---             │
│ f64        ┆ f64      ┆ f64       ┆ f64      ┆ f64      ┆ duration[μs]    │
```

[1] Эта функция позволяет вычислить расстояние между точками с учетом кривизны Земли.

```
┌──────────────┬──────────────┬────────────┬────────────┬────────────┬──────────────┐
│              │              │            │            │            │              │
│ -73.995071   │ -74.007319   │ 40.749614  │ 40.707065  │ 4.842569   │ 27m 36s 805ms │
│ -73.896576   │ -73.927311   │ 40.816459  │ 40.810893  │ 2.659582   │ 9m 25s 264ms  │
│ -73.988559   │ -73.989186   │ 40.746424  │ 40.742869  │ 0.398795   │ 3m 29s 483ms  │
│ -73.995208   │ -74.013219   │ 40.749653  │ 40.705945  │ 5.09153    │ 30m 56s 960ms │
│ -73.957559   │ -73.979881   │ 40.69067   │ 40.668663  │ 3.08728    │ 11m 32s 483ms │
│ ...          │ ...          │ ...        │ ...        │ ...        │ ...           │
│ -73.974552   │ -73.977724   │ 40.729848  │ 40.729387  │ 0.272175   │ 1m 41s 374ms  │
│ -73.971092   │ -73.965269   │ 40.763505  │ 40.763126  │ 0.492269   │ 3m 30s 363ms  │
│ -73.959621   │ -73.955151   │ 40.808625  │ 40.81      │ 0.406138   │ 1m 46s 248ms  │
│ -73.965971   │ -73.962644   │ 40.712996  │ 40.712605  │ 0.283781   │ 1m 43s 906ms  │
│ -73.940297   │ -73.940636   │ 40.707678  │ 40.703172  │ 0.501835   │ 2m 6s 109ms   │
└──────────────┴──────────────┴────────────┴────────────┴────────────┴──────────────┘
```

Обратите внимание, что расстояние между точками мы вычисляем по прямой, а не в соответствии с маршрутами поездок. Но это даст нам некое приблизительное понимание преодоленной дистанции.

## *Добавление информации об округе и районе*

Ранее мы получили координаты станций и полигоны районов. Для определения того, в каком районе располагается станция, нам нужно проверить все координаты на принадлежность полигонам. Но это не лучший способ. Некоторые станции могут не входить в полигоны, а какие-то могут иметь несколько вхождений. Все дело в расположении границ районов. Таким образом, мы можем снова воспользоваться нашим плагином, поскольку в него входит метод `Expr.geo.point_in_polygon()`:

```
stations = (
    stations.with_columns(point=pl.concat_list("lon", "lat"))
    .join(neighborhoods, how="cross")
    .with_columns(
        in_neighborhood=pl.col("point").geo.point_in_polygon(pl.col("polygon"))
    )
    .filter(pl.col("in_neighborhood"))
    .unique("station")
    .select(
        "station",
        "borough",
        "neighborhood",
    )
)

stations
```

Вывод:

```
shape: (2_133, 3)
┌──────────────────────────┬──────────┬──────────────┐
│ station                  ┆ borough  ┆ neighborhood │
│ ---                      ┆ ---      ┆ ---          │
│ str                      ┆ str      ┆ str          │
```

```
╞═══════════════════════════╪═══════════╪═══════════════╡
│ 1 Ave & E 110 St          │ Manhattan │ East Harlem   │
│ 1 Ave & E 16 St           │ Manhattan │ Stuyvesant Town │
│ 1 Ave & E 18 St           │ Manhattan │ Stuyvesant Town │
│ 1 Ave & E 30 St           │ Manhattan │ Kips Bay      │
│ 1 Ave & E 38 St           │ Manhattan │ Murray Hill   │
│ ...                       │ ...       │ ...           │
│ Wyckoff Ave & Stanhope St │ Brooklyn  │ Bushwick      │
│ Wyckoff St & 3 Ave        │ Brooklyn  │ Gowanus       │
│ Wythe Ave & Metropolitan Ave │ Brooklyn │ Williamsburg │
│ Wythe Ave & N 13 St       │ Brooklyn  │ Williamsburg  │
│ Yankee Ferry Terminal     │ Manhattan │ Governors Island │
└───────────────────────────┴───────────┴───────────────┘
```

Теперь мы можем добавить эту информацию в датафрейм `trips` путем объединения сначала со столбцом `station_start`, а затем – со столбцом `station_end`:

```
trips = (
    trips.join(
        stations.select(pl.all().name.suffix("_start")), on="station_start"
    )
    .join(stations.select(pl.all().name.suffix("_end")), on="station_end")
    .select(
        "bike_type",
        "rider_type",
        "datetime_start",
        "datetime_end",
        "duration",
        "station_start",
        "station_end",
        "neighborhood_start",
        "neighborhood_end",
        "borough_start",
        "borough_end",
        "lat_start",
        "lon_start",
        "lat_end",
        "lon_end",
        "distance",
    )
)

print(trips[:, :4])
print(trips[:, 4:7])
print(trips[:, 7:11])
print(trips[:, 11:])
```

Вывод:

```
shape: (2_638_971, 4)
┌───────────┬─────────────┬─────────────────────┬─────────────────────┐
│ bike_type ┆ rider_type  ┆ datetime_start      ┆ datetime_end        │
│ ---       ┆ ---         ┆ ---                 ┆ ---                 │
│ cat       ┆ cat         ┆ datetime[μs]        ┆ datetime[μs]        │
```

```
┌──────────┬──────────┬─────────────────────────┬─────────────────────────┐
│ electric ┆ member   ┆ 2024-03-01 00:00:02.490 ┆ 2024-03-01 00:27:39.295 │
│ electric ┆ member   ┆ 2024-03-01 00:00:04.120 ┆ 2024-03-01 00:09:29.384 │
│ ...      ┆ ...      ┆ ...                     ┆ ...                     │
│ electric ┆ member   ┆ 2024-03-31 23:55:41.173 ┆ 2024-03-31 23:57:25.079 │
│ electric ┆ member   ┆ 2024-03-31 23:57:16.025 ┆ 2024-03-31 23:59:22.134 │
└──────────┴──────────┴─────────────────────────┴─────────────────────────┘
```

```
shape: (2_638_971, 3)
```

```
┌──────────────┬──────────────────────────┬──────────────────────┐
│ duration     ┆ station_start            ┆ station_end          │
│ ---          ┆ ---                      ┆ ---                  │
│ duration[µs] ┆ str                      ┆ str                  │
╞══════════════╪══════════════════════════╪══════════════════════╡
│ 27m 36s 805ms ┆ W 30 St & 8 Ave         ┆ Maiden Ln & Pearl St │
│ 9m 25s 264ms  ┆ Longwood Ave & Southern Blvd ┆ Lincoln Ave & E 138 St │
│ ...           ┆ ...                     ┆ ...                  │
│ 1m 43s 906ms  ┆ S 4 St & Wythe Ave      ┆ S 3 St & Bedford Ave │
│ 2m 6s 109ms   ┆ Montrose Ave & Bushwick Ave ┆ Humboldt St & Varet St │
└──────────────┴──────────────────────────┴──────────────────────┘
```

```
shape: (2_638_971, 4)
```

```
┌──────────────────┬──────────────────┬──────────────┬──────────────┐
│ neighborhood_start ┆ neighborhood_end ┆ borough_start ┆ borough_end │
│ ---              ┆ ---              ┆ ---          ┆ ---          │
│ str              ┆ str              ┆ str          ┆ str          │
╞══════════════════╪══════════════════╪══════════════╪══════════════╡
│ Chelsea          ┆ Financial District ┆ Manhattan  ┆ Manhattan    │
│ Longwood         ┆ Mott Haven       ┆ Bronx        ┆ Bronx        │
│ ...              ┆ ...              ┆ ...          ┆ ...          │
│ Williamsburg     ┆ Williamsburg     ┆ Brooklyn     ┆ Brooklyn     │
│ Williamsburg     ┆ Williamsburg     ┆ Brooklyn     ┆ Brooklyn     │
└──────────────────┴──────────────────┴──────────────┴──────────────┘
```

```
shape: (2_638_971, 5)
```

```
┌───────────┬─────────────┬─────────┬─────────┬──────────┐
│ lat_start ┆ lon_start   ┆ lat_end ┆ lon_end ┆ distance │
│ ---       ┆ ---         ┆ ---     ┆ ---     ┆ ---      │
│ f64       ┆ f64         ┆ f64     ┆ f64     ┆ f64      │
╞═══════════╪═════════════╪═════════╪═════════╪══════════╡
│ 40.749614 ┆ -73.995071  ┆ 40.707065 ┆ -74.007319 ┆ 4.842569 │
│ 40.816459 ┆ -73.896576  ┆ 40.810893 ┆ -73.927311 ┆ 2.659582 │
│ ...       ┆ ...         ┆ ...     ┆ ...     ┆ ...      │
│ 40.712996 ┆ -73.965971  ┆ 40.712605 ┆ -73.962644 ┆ 0.283781 │
│ 40.707678 ┆ -73.940297  ┆ 40.703172 ┆ -73.940636 ┆ 0.501835 │
└───────────┴─────────────┴─────────┴─────────┴──────────┘
```

Перед переходом к заключительному этапу процедуры ETL давайте снова немного порисуем.

## Бонус: визуализация поездок по дням и округам

Теперь, когда у нас есть вся необходимая информация, мы можем проанализировать ее в различных разрезах. К примеру, мы можем извлечь количество поездок в разрезе дней с группировкой по округам, как показано ниже:

```
trips_per_hour = trips.group_by_dynamic(
    "datetime_start", group_by="borough_start", every="1d"
).agg(num_trips=pl.len())

trips_per_hour
```

Вывод:

```
shape: (124, 3)
```

| borough_start | datetime_start | num_trips |
|---|---|---|
| str | datetime[μs] | u32 |
| Manhattan | 2024-03-01 00:00:00 | 56434 |
| Manhattan | 2024-03-02 00:00:00 | 17450 |
| Manhattan | 2024-03-03 00:00:00 | 69195 |
| Manhattan | 2024-03-04 00:00:00 | 63734 |
| Manhattan | 2024-03-05 00:00:00 | 33309 |
| ... | ... | ... |
| Queens | 2024-03-27 00:00:00 | 6232 |
| Queens | 2024-03-28 00:00:00 | 3770 |
| Queens | 2024-03-29 00:00:00 | 6637 |
| Queens | 2024-03-30 00:00:00 | 6583 |
| Queens | 2024-03-31 00:00:00 | 6237 |

Теперь мы снова воспользуемся пакетом plotnine, чтобы создать визуализацию, показанную на рис. 1.4:

```
from mizani.labels import label_comma

(
    ggplot(
        trips_per_hour,
        aes(x="datetime_start", y="num_trips", fill="borough_start"),
    )
    + geom_area()
    + scale_fill_brewer(type="qual", palette=2)
    + scale_x_datetime(date_labels="%-d", date_breaks="1 day", expand=(0, 0))
    + scale_y_continuous(labels=label_comma(), expand=(0, 0))
    + labs(
        x="Март 2024",
        fill="Округ",
        y="Поездки по дням",
        title="Поездки на Citi Bike по дням в марте 2024 года",
        subtitle="23 марта в Нью-Йорке выпало около 10 см осадков",
    )
    + theme_tufte(base_family="Guardian Sans", base_size=14)
    + theme(
        axis_ticks_major=element_line(color="white"),
        figure_size=(8, 5),
        legend_position="top",
```

```
        plot_background=element_rect(fill="white", color="white"),
        plot_caption=element_text(style="italic"),
        plot_title=element_text(ha="left"),
    )
)
```

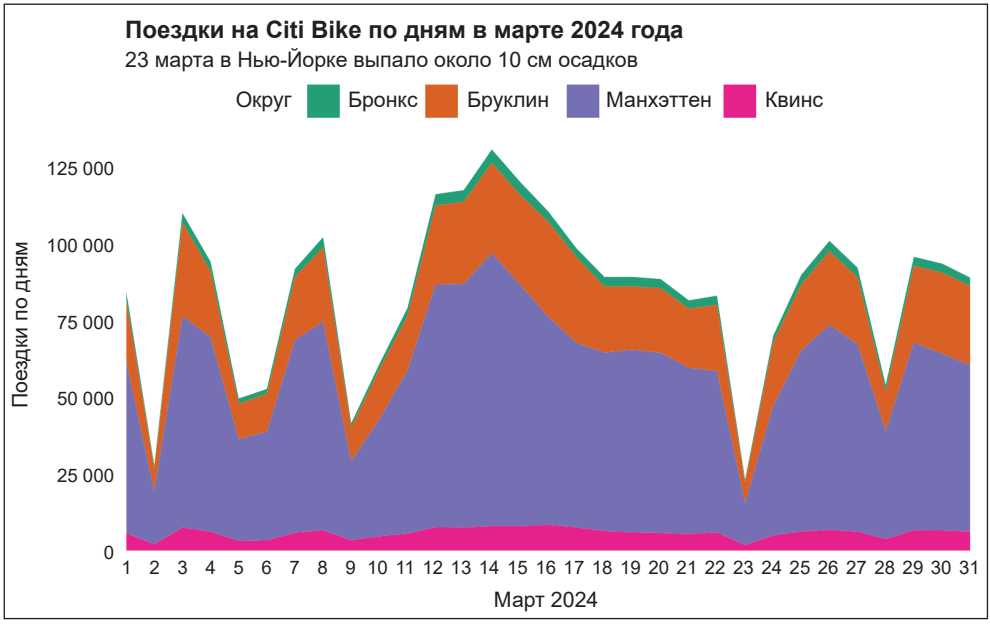



**Рис. 1.4** ❖ Поездки на Citi Bike по дням в марте 2024 года

В главе 16 мы увидим и вместе построим много разных визуализаций.

# Загрузка

Третий и заключительный шаг – процедура ETL – связан с загрузкой (*L*oad) данных. Иными словами, нам необходимо сохранить преобразованные данные на диск.

## *Запись партициями*

При записи данных на диск мы будем вместо формата CSV использовать формат *Parquet*, который обладает следующими преимуществами:

- позволяет для каждого столбца указать тип данных при помощи так называемой *схемы* (schema);
- использует колоночное хранилище вместо строчного, что позволяет быстро и оптимально осуществлять чтение;
- данные в хранилище организованы по блокам со встроенной статистикой, что позволяет пропускать ненужную информацию при чтении;

- использует сжатие данных при хранении, что снижает требования к ресурсам.

Вместо записи в один файл Parquet мы будем записывать данные по каждому дню в отдельные файлы. Это позволит сократить размер каждого файла, чтобы можно было выложить их на GitHub и поделиться с коллегами.

Имя каждого файла будет содержать слово `trips`, дефис и дату:

```
trips_parts = (
    trips.sort("datetime_start")
    .with_columns(date=pl.col("datetime_start").dt.date().cast(pl.String))
    .partition_by(["date"], as_dict=True, include_key=False)
)

for key, df in trips_parts.items():
    df.write_parquet(f"data/citibike/trips-{key[0]}.parquet")
```

## *Проверка*

Давайте проверим, что в результате выполнения этого фрагмента кода был создан 31 файл в формате Parquet, с помощью команды оболочки *`ls`*:

```
! ls -1 data/citibike/*.parquet
```

Вывод:

```
data/citibike/trips-2024-03-01.parquet
data/citibike/trips-2024-03-02.parquet
data/citibike/trips-2024-03-03.parquet
data/citibike/trips-2024-03-04.parquet
data/citibike/trips-2024-03-05.parquet
... with 26 more lines
```

Использовав символ подстановки *[1], мы можем прочитать все полученные файлы в формате Parquet в один датафрейм, как показано ниже:

```
pl.read_parquet("data/citibike/*.parquet").height
```

Вывод:

```
2638971
```

Прекрасно! Мы воспользуемся этими данными в главе 16 при создании визуализаций. Теперь, когда мы выполнили последний шаг процедуры ETL, мы можем подвести итоги этого мини-проекта. Но прежде продемонстрируем вам подход, связанный с ленивыми вычислениями, который способен существенно ускорить выполнение сценариев ETL.

---

[1] Техника подстановки применяется для выбора имен файлов по шаблону с использованием специальных символов вроде `*` и `?`.

## Бонус: ускорение за счет использования ленивых вычислений

До этого момента мы использовали только так называемый *жадный API* (eager API) Polars. Жадность в этом контексте означает, что выполнение наших команд происходит незамедлительно. Polars и в таком виде значительно превосходит конкурентов, но если воспользоваться его *ленивым API* (lazy API), вычисления можно ускорить еще больше.

Используя ленивый API, мы не оперируем с датафреймом напрямую. Вместо этого мы как бы составляем список инструкций для выполнения. А когда этот список готов, мы отдаем его Polars для запуска. Но перед непосредственным исполнением Polars сначала оптимизирует составленный нами список.

В нашем конкретном случае полный переход на ленивую парадигму был менее эффективным в сравнении с использованием жадного выполнения. Причина в том, что в этом случае нам пришлось бы дважды выполнять некоторые вычисления, поскольку Polars не знает, как кешировать результаты. В следующем фрагменте кода мы показали, как можно решить эту проблему путем частичного использования ленивой парадигмы с последующим преобразованием в обычный датафрейм для должного кеширования. Как видите, код не сильно изменился:

```
trips = (
    pl.scan_csv(
        "data/citibike/202403-citibike-tripdata.csv", ❶
        try_parse_dates=True,
        schema_overrides={
            "start_station_id": pl.String,
            "end_station_id": pl.String,
        },
    )
    .select(
        bike_type=pl.col("rideable_type").str.split("_").list.get(0),
        rider_type=pl.col("member_casual"),
        datetime_start=pl.col("started_at"),
        datetime_end=pl.col("ended_at"),
        station_start=pl.col("start_station_name"),
        station_end=pl.col("end_station_name"),
        lon_start=pl.col("start_lng"),
        lat_start=pl.col("start_lat"),
        lon_end=pl.col("end_lng"),
        lat_end=pl.col("end_lat"),
    )
    .with_columns(duration=(pl.col("datetime_end") - pl.col("datetime_start")))
    .drop_nulls()
    .filter(
        ~(
            (pl.col("station_start") == pl.col("station_end"))
            & (pl.col("duration").dt.total_seconds() < 5 * 60)
```

```
        )
    )
    .with_columns(
        distance=pl.concat_list(
            "lon_start", "lat_start"
        ).geo.haversine_distance(pl.concat_list("lon_end", "lat_end"))
        / 1000
    )
).collect() ❷

neighborhoods = (
    pl.read_json("data/citibike/nyc-neighborhoods.geojson")
    .lazy() ❸
    .select("features")
    .explode("features")
    .unnest("features")
    .unnest("properties")
    .select("neighborhood", "borough", "geometry")
    .unnest("geometry")
    .with_columns(polygon=pl.col("coordinates").list.first())
    .select("neighborhood", "borough", "polygon")
    .sort("neighborhood")
    .filter(pl.col("borough") != "Staten Island")
)

stations = (
    trips.lazy()
    .group_by(station=pl.col("station_start"))
    .agg(
        lat=pl.col("lat_start").median(),
        lon=pl.col("lon_start").median(),
    )
    .with_columns(point=pl.concat_list("lon", "lat"))
    .drop_nulls()
    .join(neighborhoods, how="cross")
    .with_columns(
        in_neighborhood=pl.col("point").geo.point_in_polygon(pl.col("polygon"))
    )
    .filter(pl.col("in_neighborhood"))
    .unique("station")
    .select(
        pl.col("station"),
        pl.col("borough"),
        pl.col("neighborhood"),
    )
).collect()

trips = (
    trips.join(
        stations.select(pl.all().name.suffix("_start")), on="station_start"
    )
    .join(stations.select(pl.all().name.suffix("_end")), on="station_end")
    .select(
```

```
        "bike_type",
        "rider_type",
        "datetime_start",
        "datetime_end",
        "duration",
        "station_start",
        "station_end",
        "neighborhood_start",
        "neighborhood_end",
        "borough_start",
        "borough_end",
        "lat_start",
        "lon_start",
        "lat_end",
        "lon_end",
        "distance",
    )
)

trips.height
```

❶ Функция *scan_csv()* возвращает ленивый датафрейм, или *LazyFrame*, что переводит выполнение последующих методов в парадигму ленивых вычислений.
❷ Метод *collect()* преобразует LazyFrame в DataFrame.
❸ Метод *lazy()* преобразует DataFrame обратно в LazyFrame.

Вывод:

```
2639179
```

При анализе одного месяца поездок на велосипедах использование библиотеки Polars не сильно помогло нам в плане эффективности, поскольку львиную долю времени заняло определение вхождения точек в полигоны. Но если попробовать проанализировать поездки за год, ленивый подход даст прибавку в 33 % над жадным. Неплохое улучшение всего для нескольких изменений в коде.

Подробнее о жадном и ленивом API мы будем говорить в главе 5.

# Заключение

В этой главе вы узнали, что:

- Polars – это библиотека, предназначенная для быстрой и интуитивно понятной обработки табличных данных в виде датафреймов;
- библиотека Polars написана на языке Rust и имеет языковые привязки к Python, R, SQL, JavaScript и Julia;
- API для Python является наиболее совершенным и часто используемым интерфейсом Polars;

- Polars – действительно очень популярный и набирающий обороты пакет в Python, о чем говорит динамика звезд на GitHub;
- Polars в большинстве случаев опережает в плане быстродействия своих конкурентов;
- при использовании ленивого API Polars может показывать еще большее быстродействие;
- библиотека Polars отлично подходит для преобразования, анализа и визуализации данных.

В следующей главе мы подробно поговорим о процессе установки библиотеки Polars и ее использовании. Также мы расскажем, как вы можете запускать на выполнение все примеры из этой книги.

# Глава 2

# Установка и начало работы

Чтобы воспользоваться всеми благами библиотеки Polars, для начала необходимо установить ее и подготовить к работе.

В этой главе вы узнаете:

- как настроить свое рабочее окружение;
- как установить необходимые зависимости в созданное виртуальное окружение с помощью инструмента *uv* для работы с примерами из этой книги;
- как запускать и использовать JupyterLab – оболочку, в которой будете работать с примерами на Polars;
- как установить библиотеку Polars и необходимые для работы зависимости;
- как сконфигурировать Polars под ваши собственные нужды;
- как скомпилировать Polars из исходников и что делать, если вы работаете с очень большими наборами данных или используете более старые процессоры.

Мы настоятельно рекомендуем вам самостоятельно проверять все примеры из этой книги. Изучение новых пакетов происходит гораздо эффективнее, когда вы пробуете все на практике, а не просто читаете о том, что можно сделать.

## Настройка рабочего окружения

Для проверки всех примеров из этой книги вам понадобится настроить окружение. Сначала мы загрузим все файлы проекта, после чего установим утилиту *uv*, требующуюся для создания и использования окружения. Это очень эффективный менеджер пакетов и проектов на Python, написанный на языке Rust. С помощью этого инструмента вы легко установите все зависимости, необходимые для запуска примеров из этой книги.

# Загрузка файлов проекта

Загрузить файлы проекта можно двумя разными способами. Первый состоит в установке Git, а второй предполагает прямое скачивание архива:

- с помощью Git: мы рекомендуем использовать этот подход, чтобы можно было автоматически синхронизировать проект с сервером. Инструкции по установке Git можно почитать по адресу *https://git-scm.com/book/en/v2/Getting-Started-Installing-Git*. После установки Git вам достаточно будет ввести следующую команду в терминале:

```
$ git clone https://github.com/jeroenjanssens/python-polars-the-definitive-guide
```

- путем скачивания архива: вы можете вручную загрузить архив с файлами проекта из репозитория на GitHub, нажав на зеленую кнопку **Code**. В открывшемся окне нажмите на кнопку **Download ZIP**, как показано на рис. 2.1. После загрузки архива вы можете распаковать его в любую директорию по выбору.

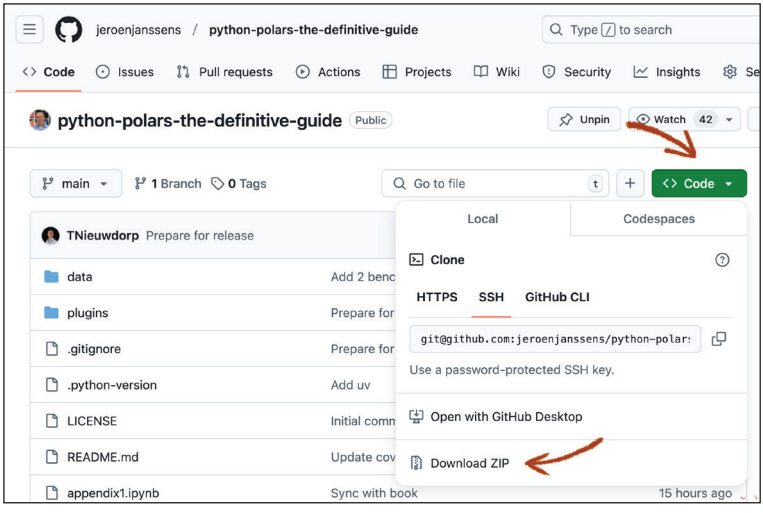



**Рис. 2.1** ❖ Кнопка загрузки архива на GitHub

Теперь файлы проекта располагаются на вашем компьютере, и мы можем приступать к созданию рабочего окружения. Для этого воспользуемся утилитой *uv*.

# Установка утилиты uv

Полную информацию по установке утилиты *uv* вы можете получить по адресу *https://docs.astral.sh/uv/getting-started/installation*.

Мы рекомендуем использовать автономный установщик, который можно инсталлировать следующими способами:

- на Linux и macOS вы можете установить *uv* путем выполнения показанной ниже команды в любом удобном вам терминале:

```
$ curl -LsSf https://astral.sh/uv/install.sh | sh
```

- если вы используете Windows, установить утилиту *uv* можно с помощью следующей команды в PowerShell:

```
PS C:\Users\You> powershell -ExecutionPolicy ByPass -c
"irm https://astral.sh/uv/install.ps1 | iex"
```

**Доллары, восклицательные знаки и проценты**

Вы заметили, что некоторые строки во фрагментах кода начинаются со знака доллара ($) или `PS >`. Это не часть команды, а строка приглашения, после которой необходимо вводить сами команды в терминале Linux или macOS (в случае с $) или Windows PowerShell (в случае `PS >`). *Не* вводите строку приглашения вручную. Если вы видите перед фрагментом кода восклицательный знак (!) или знак процента (%), это означает, что эти команды необходимо вводить в Jupyter Notebook, и эти символы нужно вводить вручную.

# Установка проекта

После установки утилиты *uv* можно легко и просто запустить проект следующей командой:

```
$ uv sync
```

# Работа в виртуальном окружении

В процессе запуска *uv* будет создано *виртуальное окружение* (virtual environment), в котором вы устанавливаете все нужные зависимости для запуска примеров из этой книги. Виртуальное окружение представляет собой отдельную директорию, содержащую установленный интерпретатор Python определенной версии и все необходимые дополнительные пакеты. Таким образом, вы можете работать со множеством проектов, каждый из которых предусматривает наличие своих пакетов. Существует два способа работы с виртуальными окружениями:

- рекомендованный способ состоит в добавлении префикса `uv run` к своим командам, например `uv run jupyter lab` можно использовать для запуска Jupyter с использованием виртуального окружения. Этот способ будет работать на всех операционных системах;
- второй способ состоит в активировании виртуального окружения и запуске последующих команд в обычном режиме:

– в macOS и Linux активировать виртуальное окружение можно следующей командой:

```
$ source .venv/bin/activate
```

– в Windows для активации виртуального окружения используется такая команда:

```
PS > .venv\Scripts\activate
```

При возвращении к работе над проектом вам необходимо ввести одну из указанных команд, чтобы активировать требуемое виртуальное окружение.

# Проверка установки

На протяжении книги мы будем использовать библиотеку Polars версии 1.20.0. Запустите приведенный ниже код в новом ноутбуке, чтобы проверить версию используемой библиотеки:

```
import polars as pl

pl.show_versions()
```

Вывод:

```
--------Version info---------
Polars:              1.20.0
Index type:          UInt32
Platform:            macOS-14.5-arm64-arm-64bit
Python:              3.12.6 (main, Sep  9 2024, 21:36:32) [Clang 18.1.8 ]
LTS CPU:             False
----Optional dependencies----
Azure CLI            2.67.0
adbc_driver_manager  1.2.0
altair               5.4.1
azure.identity       <not installed>
boto3                <not installed>
cloudpickle          3.1.0
connectorx           0.4.0
deltalake            0.21.0
fastexcel            0.12.0
fsspec               2024.10.0
gevent               24.11.1
google.auth          <not installed>
great_tables         0.14.0
matplotlib           3.9.2
nest_asyncio         1.6.0
numpy                2.1.3
openpyxl             3.1.5
pandas               2.2.3
```

```
pyarrow              18.0.0
pydantic             2.9.2
pyiceberg            0.7.1
sqlalchemy           2.0.36
torch                <not installed>
xlsx2csv             0.8.3
xlsxwriter           3.2.0
```

# Экспресс-курс по JupyterLab

Для выполнения примеров из книги вам потребуется Jupyter. *Jupyter* – это интерактивная веб-среда разработки на основе ноутбуков, кода и данных. Для начала работы с Polars вам достаточно открыть ноутбук, относящийся к любой главе этой книги, в меню слева или создать свой собственный ноутбук, нажав на кнопку **Python 3 Notebook**.

Для запуска Jupyter вы можете ввести следующую команду в терминале:

```
$ uv run jupyter lab
```

В результате в браузере откроется окно с интерфейсом Jupyter. Если окно само не откроется, вы можете вручную вставить в адресную строку браузера ссылку из терминала. Она должна выглядеть как-то так: *http://127.0.0.1:8888/lab?token=...*. Как итог, вы подключитесь к серверу Jupyter внутри контейнера и сможете с ним работать.

Для работы с ноутбуками в Jupyter вам необходимо знать всего несколько базовых правил. Среда Jupyter содержит ячейки. Эти ячейки могут использоваться как для кода на разных языках программирования, так и для создания оформления в виде разметки текста (Markdown). В этой книге в большинстве случаев вы будете работать с ячейками с кодом на языке Python.

Для навигации по ячейкам в Jupyter применяется два режима: *режим управления* (command mode) и *режим редактирования* (edit mode):

- режим управления активируется по умолчанию при открытии ноутбука или при нажатии на клавишу **Esc** в ячейке. В этом режиме выбранная ячейка обводится синей границей, а курсор в ней отсутствует. Режим управления используется для редактирования ноутбука в целом, а также для добавления, редактирования и удаления ячеек разных типов;
- режим редактирования активируется по нажатии клавиши **Enter** на выбранной ячейке. Он предназначается для того, чтобы в ячейке, обведенной зеленой границей, можно было печатать код или текст.

## Горячие клавиши в Jupyter

Есть несколько важных горячих клавиш, которые вам необходимо знать для комфортной работы с Jupyter. В табл. 2.1 перечислены сочетания клавиш,

которые можно использовать во всех режимах, в табл. 2.2 – сочетания для режима управления, а в табл. 2.3 – для режима редактирования.

## Все режимы

Сочетания клавиш, перечисленные в табл. 2.1, могут быть использованы всегда, вне зависимости от текущего режима.

*Таблица 2.1. Универсальные горячие клавиши*

| Сочетание клавиш | Действие |
|---|---|
| **Shift**+**Enter** | Выполнить текущую ячейку и перейти в следующую |
| **Ctrl**+**Enter** | Выполнить текущую ячейку и не переходить в следующую |
| **Alt**+**Enter** | Выполнить текущую ячейку и вставить пустую ячейку ниже |
| **Ctrl**+**S** | Сохранить ноутбук |

## Режим управления

Сочетания клавиш из табл. 2.2 могут быть использованы только в режиме управления.

*Таблица 2.2. Горячие клавиши для режима управления*

| Сочетание клавиш | Действие |
|---|---|
| **Enter** | Переключиться в режим редактирования |
| **Up**/**K** | Выбрать ячейку выше |
| **Down**/**J** | Выбрать ячейку ниже |
| **A** | Вставить новую ячейку над текущей |
| **B** | Вставить новую ячейку под текущей |
| **D**, **D** (двойное нажатие) | Удалить текущую ячейку |
| **Z** | Отменить удаление ячейки |
| **M** | Изменить тип ячейки на Markdown |
| **Y** | Изменить тип ячейки на Code |

## Режим редактирования

Сочетания клавиш, перечисленные в табл. 2.3, могут быть использованы только в режиме редактирования

*Таблица 2.3. Горячие клавиши для режима редактирования*

| Сочетание клавиш | Действие |
|---|---|
| **Esc** | Переключиться в режим управления |
| **Ctrl**+**Shift**+- | Разделить текущую ячейку по курсору |

Если вы будете активно использовать эти горячие клавиши, то будете буквально летать по ноутбуку Jupyter, не теряя много времени.

Также при работе с ноутбуками можно использовать специальные символы прямо в ячейках для изменения контекста запуска. К примеру, восклицательный знак (!) перед строкой кода указывает Jupyter на необходимость запустить ее в терминале, а не перенаправлять интерпретатору Python. К примеру, вы можете посмотреть содержимое текущей папки с помощью следующей простой команды:

```
! ls
```

Еще одним специальным символом является знак процента (%). Он служит для вызова так называемых *магических команд* (magic commands) движка IPython. Это команды, предназначенные для выполнения определенных задач и часто помогающие разработчику. Они не являются частью языка Python, а входят в состав оболочки *IPython*. Магические команды делятся на два следующих типа:

- магические команды уровня строки: такие команды предваряются одиночным символом процента (*%*) и работают так же, как команды оболочки. К примеру, вы можете воспользоваться магической командой *%cat* для отображения содержимого файла;
- магические команды уровня ячейки: перед такими командами ставится два символа процента (*%%*). Примером является команда *%%time*, позволяющая засечь время, в течение которого выполнялась ячейка. Также можно вспомнить команду *%%bash*, дающую возможность запустить несколько команд в терминале одновременно.

Для просмотра полного списка магических команд оболочки IPython можно воспользоваться командой *%lsmagic*.

**Jupyter Notebook против JupyterLab**

*JupyterLab* представляет собой последнюю разработку интерактивной веб-среды на основе ноутбуков, кода и данных. Изначально Jupyter основывался исключительно на отдельных ноутбуках, но позже над этим уровнем была возведена надстройка, получившая имя JupyterLab. С помощью нее можно удобно работать с документами и другими сущностями среды, такими как ноутбуки, текстовые редакторы, терминалы и пользовательские компоненты на основе браузерного интерфейса. В этой книге мы будем пользоваться средой JupyterLab для запуска примеров кода.

# Установка Polars в другие проекты

Хотя мы уже настроили наше рабочее окружение, вам будет полезно узнать, как устанавливать Polars в собственные проекты. Актуальную информацию по установке библиотеки и всех ее зависимостей можно найти на GitHub по адресу *https://github.com/pola-rs/polars* и в инструкции на официальном сайте по адресу *https://docs.pola.rs/user-guide/installation*. Именно на этих документах мы базировались, когда писали это руководство.

Библиотека Polars может быть установлена с разными наборами зависимостей, требующихся для разных проектов. Эти дополнительные зависимости объединены в группы, о которых мы сейчас кратко поговорим.

## Дополнительные зависимости

Все возможные опциональные зависимости могут быть установлены с библиотекой Polars с использованием нотации на основе квадратных скобок. Для установки последней версии Polars со всеми зависимостями можно выполнить следующую команду в терминале:

```
$ uv pip install 'polars[all]'
```

**Не uv единым...**

Как видите, мы в этой книге в основном используем утилиту командной строки *uv*. Дело в том, что мы считаем *uv* наиболее быстрым и эргономичным инструментом для управления окружениями в Python. Если желаете, можете продолжать использовать привычные для вас утилиты *pip* и *conda*. К примеру, для установки Polars с помощью pip вы можете ввести следующую команду с тегом `all`:

```
$ pip install 'polars[all]'
```

Команда установки с использованием утилиты conda будет выглядеть так:

```
$ conda install -c conda-forge polars
```

## Зависимости для совместимости

Для взаимодействия с другими пакетами для работы с датафреймами вы можете опционально установить перечисленные в табл. 2.4 зависимости.

***Таблица 2.4. Опциональные зависимости для взаимодействия с другими пакетами обработки данных***

| Тег | Описание |
|---|---|
| `Pandas` | Преобразование данных из/в формат датафреймов/Series в pandas |
| `Numpy` | Преобразование данных из/в формат массивов NumPy |
| `Pyarrow` | Преобразование данных из/в формат таблиц/массивов PyArrow |
| `Pydantic` | Преобразование данных из моделей Pydantic в Polars |

## Зависимости для работы с электронными таблицами

Установка зависимостей, перечисленных в табл. 2.5, позволит вам работать с Excel и другими форматами электронных таблиц. Вы можете установить их при помощи тега `excel`.

*Таблица 2.5. Опциональные зависимости для работы с электронными таблицами*

| Тег | Описание |
|---|---|
| `Excel` | Установка *всех* движков для работы с Excel |
| `Calamine` | Чтение из файлов Excel с помощью движка calamine |
| `Openpyxl` | Чтение из файлов Excel с помощью движка openpyxl |
| `xlsx2csv` | Чтение из файлов Excel с помощью движка xlsx2csv |
| `Xlsxwriter` | Запись в файлы Excel с помощью движка XlsxWriter |

## Зависимости для работы с базами данных

Установка зависимостей из табл. 2.6 даст вам возможность подключаться к базам данных и загружать из них нужную информацию. Все зависимости сразу можно установить при помощи тега `database`.

*Таблица 2.6. Опциональные зависимости для работы с базами данных*

| Тег | Описание |
|---|---|
| `Database` | Установка *всех* движков для работы с базами данных |
| `Adbc` | Чтение и запись в базы данных с помощью движка Arrow Database Connectivity (ADBC) |
| `connectorx` | Чтение из баз данных с помощью движка ConnectorX |
| `sqlalchemy` | Запись в базы данных с помощью движка SQLAlchemy |

## Зависимости для работы с удаленными файловыми системами

Если у вас есть необходимость подключения к удаленной файловой системе, Polars позволит это сделать с помощью пакета *`fsspec`*, приведенного в табл. 2.7.

*Таблица 2.7. Опциональные зависимости для работы с облачными платформами*

| Тег | Описание |
|---|---|
| `Fsspec` | Чтение и запись в удаленные файловые системы |

## Зависимости для работы с другими форматами ввода/вывода

В мире существует немало форматов ввода/вывода, и Polars может взаимодействовать с ними посредством зависимостей, перечисленных в табл. 2.8.

***Таблица 2.8. Опциональные зависимости для работы с другими форматами ввода/вывода***

| Тег | Описание |
|---|---|
| `Deltalake` | Чтение и запись в таблицы Delta |
| `Iceberg` | Чтение из таблиц Apache Iceberg |

# Зависимости для использования расширенного функционала

Помимо чтения и записи данных, Polars также поддерживает несколько зависимостей, обеспечивающих ему дополнительный функционал. Эти зависимости перечислены в табл. 2.9.

***Таблица 2.9. Опциональные зависимости для использования расширенного функционала***

| Тег | Описание |
|---|---|
| `async` | Асинхронный сбор ленивых датафреймов |
| `cloudpickle` | Расширенный функционал pickle для сериализации пользовательских функций |
| `graph` | Визуализация ленивых датафреймов в виде графов |
| `plot` | Графическое отображение датафреймов с помощью пространства имен `plot` |
| `style` | Стилизация датафреймов с помощью пространства имен `style` |
| `timezone` | Поддержка временных зон[1] |

# Установка дополнительных зависимостей

Если вам необходимо установить некое подмножество дополнительных зависимостей, вы можете воспользоваться следующим синтаксисом:

```
$ uv pip install 'polars[pandas,numpy]'
```

Если вам необходимо установить только базовый пакет, легче всего это сделать с помощью простой команды:

```
$ uv pip install polars
```

[1] Поддержка временных зон может понадобиться, если вы работаете с Python < 3.9 на Windows.

# Конфигурирование Polars

В большинстве случаев библиотека Polars прекрасно работает «из коробки», т. е. в базовой комплектации. Так что дополнительные настройки вам могут не понадобиться вовсе. Но никогда нелишним будет узнать, какие настройки вообще возможны.

Polars предлагает большое количество опций конфигурации. С помощью них можно менять форматирование выходных таблиц, задавать уровни логирования и устанавливать размер блока в потоке. В классе *polars.Config* можно найти настройки, показанные в табл. 2.10, и многие другие. Полный список настроек можно посмотреть в официальной документации библиотеки по адресу *https://docs.pola.rs/api/python/stable/reference/config.html*.

***Таблица 2.10. Некоторые наиболее популярные настройки библиотеки Polars***

| Метод | Описание |
|---|---|
| `pl.Config.set_decimal_separator(...)` | Установка символа разделителя десятичных знаков |
| `pl.Config.set_float_precision(...)` | Установка количества знаков после запятой при отображении вещественных чисел |
| `pl.Config.set_fmt_str_lengths(...)` | Установка количества символов для отображения строковых значений |
| `pl.Config.set_tbl_cols(...)` | Установка количества столбцов, которые будут видимы при выводе таблицы |
| `pl.Config.set_tbl_formatting(...)` | Установка стиля форматирования таблиц |
| `pl.Config.set_tbl_rows(...)` | Установка максимального количества строк, которые будут видимы при выводе таблицы (для датафреймов и объектов Series) |
| `pl.Config.set_tbl_width_chars(...)` | Установка максимальной ширины столбца в символах |
| `pl.Config.set_thousands_separator(...)` | Установка символа разделителя разрядов |
| `pl.Config.set_verbose(...)` | Установка уровня логирования/отладки |

Эти опции конфигурации можно менять, сохранять и загружать в виде строки JSON или файла с помощью функций *pl.Config.load()*, *pl.Config.load_from_file()*, *pl.Config.save()* и *pl.Config.save_to_file()*. Для просмотра текущего состояния настройки можно воспользоваться функцией *pl.Config.state()*. Для установки всех настроек в состояние по умолчанию можно вызвать функцию *pl.Config.restore_defaults()*.

## Временная конфигурация с использованием контекстного менеджера

Для запуска конкретного фрагмента кода с измененной конфигурацией можно воспользоваться *контекстным менеджером* (context manager). Контекстный менеджер представляет собой конструкцию в Python, позволяющую локально создавать и удалять ресурсы. Контекст, для которого определяются

ресурсы, создается при помощи ключевого слова *with* с отступом, который и ограничивает контекст. Таким образом, только код, находящийся внутри созданного контекста, будет запущен с измененной конфигурацией, а после выхода из контекста будут возвращены прежние настройки:

```
with pl.Config() as cfg:
    cfg.set_verbose(True)
    # Операции в Polars с измененными настройками

# Код за пределами контекста будет выполняться с прежними настройками
```

Более лаконичный подход состоит в передаче настроек непосредственно в конструктор `Config()`. В этом случае вам не нужно будет вызывать методы с префиксом `set_`, как показано ниже:

```
with pl.Config(verbose=True):
    # Операции Polars с многословным логированием
    pass
```

Для проверки опций, связанных с форматированием, нам сначала придется создать датафрейм. Датафрейм – это двумерная структура данных, представляющая содержимое в виде таблицы, состоящей из строк и столбцов. И это основная структура данных, используемая в Polars. Позже в этой книге мы познакомим вас со всеми структурами библиотеки.

В следующем фрагменте кода мы создали простую функцию, генерирующую случайную строку заданной длины. Далее мы наполнили словарь десятью ключами (от `column_1` до `column_10`) и заполнили значения пятью случайно сгенерированными строками длины 50, объединенными в список. В завершение мы создали на основе этого словаря датафрейм:

```
import random
import string

def generate_random_string(length: int) -> str:
    return "".join(random.choice(string.ascii_letters) for i in range(length))

data = {}
for i in range(1, 11):
    data[f"column_{i}"] = [generate_random_string(50) for _ in range(5)] ❶

df = pl.DataFrame(data)
```

❶ Здесь мы для создания имен столбцов используем f-строки. *F-строки* (f-string) позволяют очень комфортно работать со строками, и они появились в Python еще в версии 3.6.

Посмотрим, как выглядит наш датафрейм:

```
df
```

Вывод:

```
shape: (5, 10)
```

| column_1 | column_2 | column_3 | column_4 | … | column_7 | column_8 | column_9 | column_10 |
|---|---|---|---|---|---|---|---|---|
| str | str | str | str | | str | str | str | str |
| VWIUPmaZl | FzsxjbIpI | GbiABZuLR | KNUblmmBS | … | NNArtrqBp | lvBkoFxIz | OsyNtKhYq | LMBunCpR |
| vgCqSqGkt | wEgIKvLpy | wdsArnWsw | rOGVsxArb | | RrAKvAewe | zzcQdUlEt | XOGePnJLa | JdtWRlmD |
| eeIClGMyu | PWqwimQQL | OCVZsRsSO | LvqSwFNsv | | QzalPDoNR | UhZhJIivj | WJEuVCwso | PJuTCOEg |
| jxe… | FFu… | Lwx… | CKb… | | DfI… | lOw… | xMt… | JWXpLA… |
| QeHndKion | tFRYQVVGV | NoFSzIszu | OmHHhlTrU | … | ApNoXGeYR | nudIZkaRP | XMITmJJFQ | orrUxaFD |
| XJkqKWgaB | TUwRrPXMW | WQOLghCZm | ivekMbMfY | | kTOrAUALq | ovROauIMV | XuWjFDlvM | VbLGQOPT |
| AWGlcOpBp | enSpKoHEk | NWUpyulft | xVPugssJJ | | PexghorHi | OUeGamlLM | HEMIGUaXw | bFOeDtSw |
| spw… | elE… | obc… | DfW… | | mGr… | vci… | rIG… | tjeVyn… |
| ftLGRDDSf | AYVYFZjPZ | JRPZeZtcA | XyBngpYKa | … | nqDFNJNwm | argHzTQeU | KtdMrzOUw | MLMsPUAc |
| hUCnPuLoE | EqVPWhLSd | VZepjveKD | KxoKyTZXr | | rbeWaRmlU | MOWYCOqDC | BGUbTTHFb | SRKUEWDb |
| tnyYGDygL | vcOuSJwmB | QBSPpryBE | JSgiSTPBN | | kqkSioEKc | efnKwDMde | EvHLUfBfo | nzmAcmWC |
| syV… | hKn… | rNR… | sbu… | | eWY… | DCW… | Dch… | NIeJkI… |
| TAjTNVIdQ | dEltgKQky | usgIntkCX | DNJDXEUTi | … | hFvLpZZTj | LWHqBMFtN | GPPeTPUuj | oETCnvkI |
| uQHJjNzhr | bcZBESCZB | QFrezUuAC | QPaOcHSfy | | WJMSZjzws | CncvhqAQw | HxyFbJWjj | mCvkrQxr |
| TjePlaSiW | MxNtisMrG | eoDmGtOqs | JdckjKnuN | | EBDtBYWWP | NWKCokcIg | CFLyIzPsj | ZgjAyBzC |
| DKt… | ldk… | YCB… | mwm… | | SYj… | QoY… | cLt… | cuYSsx… |
| CKCfEqwpZ | akHHORQcs | oDabeZRgZ | osOlijiWA | … | FbmqpjzwK | CsReepFEu | QaDmVfGYl | crrxVUxx |
| YLXmbUfkW | YwqbIhYdF | purzaAQzk | dXzLNTCvz | | mSnQHXTzz | nIwxghfXe | UGYUAOVvs | njuLcoca |
| ygGvvRYyO | rhqfLlztY | HrUizgyEY | BccaQKMMa | | KNanNKFQu | iMjBKyIDt | ZWMyMhXeC | VMFzLLyt |
| RMm… | QlQ… | Yek… | met… | | Owd… | Fjs… | dfo… | njeRUh… |

Как видите, не все колонки поместились в вывод. А что делать, если вам гарантированно нужно увидеть их все, жертвуя при этом длиной отображаемого содержимого? В этом случае вы можете задать настройку для вывода всех столбцов (`tbl_cols=-1`) и снизить длину отображаемых строк до четырех символов (`fmt_str_lengths=4`), как показано ниже:

```
with pl.Config(tbl_cols=-1, fmt_str_lengths=4):
    print(df)
```

Вывод:

```
shape: (5, 10)
```

| colu… | colu… | colu… | colu… | colu… | colu… | colu… | colu… | colu… | colu… |
|---|---|---|---|---|---|---|---|---|---|
| str | str | str | str | str | str | str | str | str | str |
| VWIU… | Fzsx… | GbiA… | KNUb… | vUlb… | uXVk… | NNAr… | lvBk… | OsyN… | LMBu… |
| QeHn… | tFRY… | NoFS… | OmHH… | dnxU… | BHhA… | ApNo… | nudI… | XMIT… | orrU… |

```
| ftLG… | AYVY… | JRPZ… | XyBn… | UXtv… | RXEo… | nqDF… | argH… | KtdM… | MLMs… |
| TAjT… | dElt… | usgI… | DNJD… | VADO… | BEcl… | hFvL… | LWHq… | GPPe… | oETC… |
| CKCf… | akHH… | oDab… | osOl… | TBdS… | giXr… | Fbmq… | CsRe… | QaDm… | crrx… |
└───────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┘
```

Теперь все колонки поместились в вывод, пусть и в сокращенном виде.

**И это все о контексте**

Контекстные менеджеры содержат под капотом два ключевых метода *__enter__()* и *__exit__()*, которые вызываются перед запуском фрагмента кода и по его окончании соответственно. Ниже показан небольшой пример:

```
class YourContextManager:
    def __enter__(self):
        print("Запуск контекста")

    def __exit__(self, type, value, traceback):
        print("Выход из контекста")

with YourContextManager():
    print("Ваш код...")
```

Вывод:

```
Запуск контекста
Ваш код...
Выход из контекста
```

Один из распространенных способов применения контекстных менеджеров состоит в чтении файлов, как показано ниже:

```
with open("data/fruit.csv", "r") as file:
    print(file.readline())
```

# Локальная конфигурация с использованием декораторов

Если вы хотите вызвать конкретную функцию с измененными настройками, вы можете воспользоваться *декоратором* (decorator) `pl.Config()`. Как и в случае с передачей настроек в виде параметра в контекстный менеджер, в этом случае вам не придется вызывать методы с префиксом `set_`:

```
@pl.Config(ascii_tables=True)
def write_ascii_frame_to_stdout(df: pl.DataFrame) -> None:
    print(str(df))

@pl.Config(verbose=True)
def function_that_im_debugging(df: pl.DataFrame) -> None:
    # Операции Polars с многословным логированием
    pass
```

# Компилирование Polars с нуля

Вариант с компилированием Polars из исходных файлов имеет ряд преимуществ. Хотя напрямую к Polars это не имеет отношения, поскольку новые релизы данной библиотеки выпускаются достаточно регулярно, в общем случае компиляция с нуля позволяет получить более оперативный доступ к последним изменениям. Также этот способ дает вам возможность внести изменения в исходный код, самостоятельно скомпилировать его и воспользоваться добавленным функционалом. А если ваше нововведение может быть полезным для всех, вы можете добавить его в ваш проект.

При работе с нестандартной архитектурой компьютера самостоятельное компилирование кода бывает даже необходимой мерой, поскольку готового скомпилированного пакета под ваши требования может просто не оказаться. Кроме того, если вы обладаете достаточными знаниями, вы можете оптимизировать процесс компиляции, что позволит создать более быструю и эффективную версию пакета.

Действия, необходимые для компиляции библиотеки Polars из исходников, перечислены ниже.

1. Установить компилятор Rust, воспользовавшись инструкцией на странице *https://www.rust-lang.org/tools/install*.
2. Установить пакет нулевой конфигурации *maturin*, помогающий осуществлять сборку и публикацию крейтов на Rust с привязкой к Python. Вы можете сделать это в терминале с помощью следующей команды:

   ```
   $ uv pip install maturin
   ```

   Важно, что вы должны установить этот пакет в виртуальное окружение проекта, с которым работаете.
3. Скомпилировать двоичный файл. Это можно сделать двумя способами:
   – если для вас важнее быстродействие пакета, а не время компиляции (к примеру, если вы готовы скомпилировать его однократно и затем использовать), вы можете воспользоваться следующей командой:

     ```
     $ maturin develop --release -m py-polars/Cargo.toml --
      -C target-cpu=native
     ```

   – если у вас в приоритете время сборки пакета (например, если вы тестируете изменения), то можете выбрать следующую команду:

     ```
     $ maturin develop --release -m py-polars/Cargo.toml --
      -C codegen-units=16 -C lto=thin -C target-cpu=native
     ```

**Имена крейта, пакета и модуля**

Крейт в Rust, реализующий привязку к Python, называется *py-polars*, чтобы можно было отличать его от родного крейта Polars в Rust. Однако поскольку и пакет, и модуль в Python именуются *polars*, вы можете использовать инструкции вроде `uv pip install polars` и `import polars`.

## Особый случай: работа с очень большими наборами данных

Если вы работаете с действительно объемными наборами данных, количество строк в которых превышает 4.2 млрд, вам потребуется установить Polars несколько иначе. Для отслеживания хранящихся данных Polars по умолчанию пользуется 32-битным целочисленным представлением. При работе с большими данными вам может понадобиться скомпилировать Polars с флагом *bigidx* для расширения внутренних представлений. Также эту установку Polars можно выполнить с помощью инструкции `uv pip install polars-u64-idx`. Это может привести к некоторой потере быстродействия, если вам в действительности не нужна поддержка больших наборов данных.

## Особый случай: процессоры без поддержки AVX

*AVX* (Advanced Vector Extensions) представляет собой набор инструкций, разработанных компанией Intel, который предназначен для ускорения выполнения операций с векторными данными. Эти инструкции позволяют центральному процессору более быстро и эффективно производить сложные вычисления. Впервые набор инструкций AVX был реализован в процессорах Intel и AMD в 2011 году. На более ранних процессорах эти технологии, увы, не реализованы. Если ваш чипсет не поддерживает AVX, вам придется установить пакет *polars-lts-cpu*. Его можно найти в репозитории PyPI или установить при помощи команды `uv pip install polars-lts-cpu`.

Одновременная установка пакетов polars и polars-lts-cpu может привести к появлению ошибок при импорте и использовании библиотеки Polars.

# Заключение

В этой главе вы узнали:

- как настроить свое рабочее окружение при помощи uv и Git;
- как запускать примеры кода в JupyterLab;
- как установить Polars и его дополнительные зависимости;
- как на лету менять конфигурацию Polars при выполнении кода;
- как и зачем может понадобиться компилировать Polars из исходников.

Все это позволяет вам установить Polars в нужной для вас конфигурации и начать работать с библиотекой. В следующей главе мы подробно поговорим про сходства и различия между Polars и наиболее популярной на данный момент библиотекой для работы с табличными данными pandas.

# Глава 3

# От pandas к Polars

Мы отдаем себе отчет в том, что читатели этой книги, скорее всего, хорошо знакомы с библиотекой pandas, предназначенной для анализа и манипулирования данными в табличном виде. Если так случилось, что вы никогда не работали с pandas, вы можете просто пропустить эту главу. Но если у вас есть опыт работы с pandas, вам будет очень полезно ознакомиться с тем, что мы напишем, – это поможет вам в освоении дальнейшего материала книги.

Итак, основной целью этой главы является обеспечение наиболее плавного, насколько это возможно, перехода от pandas к Polars, и для этого мы скрупулезно пройдемся по всем сходствам и, что более важно, различиям между этими библиотеками.

С момента своего появления библиотека Polars находилась под пристальным испытующим взором со стороны апологетов pandas. И хотя объективно сравнение зачастую было в пользу Polars, далеко не все были с этим согласны. И это вполне понятно. Причина в том, что специалисты по работе с данными во многом обязаны pandas. Более того, без pandas, скорее всего, не было бы и Polars. Да даже больше – если бы не pandas, язык Python вряд ли вообще стал бы таким популярным в сфере обработки и анализа данных. В этой главе мы постараемся максимально объективно описать все сходства и различия этих двух библиотек.

В частности, вы узнаете:

- что общего у библиотек pandas и Polars;
- чем отличается синтаксис и вывод инструкций обеих библиотек;
- какие концепции, применяемые в pandas, вам придется забыть;
- как в обеих библиотеках выполняются наиболее распространенные операции.

Здесь мы главным образом будем говорить о пользовательском опыте, а не о внутренней механике процессов. Кроме того, в этой главе мы не будем сравнивать быстродействие pandas и Polars. Все инструкции по загрузке и установке требуемых файлов вы найдете в главе 2. Мы будем исходить из предположения о том, что все рабочие файлы вы поместили в директорию *data*.

# Набор данных с животными

В этой главе мы будем работать с файлом *data/animals.csv*, содержащим информацию о десятке видов животных и их свойствах, таких как среда обитания, средняя продолжительность жизни и вес. Это совсем небольшой набор данных, но для целей данной главы его будет вполне достаточно. Сырые данные выглядят так:

```
! cat data/animals.csv
```

Вывод:

```
animal,class,habitat,diet,lifespan,status,features,weight
dolphin,mammal,oceans/rivers,carnivore,40,least concern,high intelligence,150
duck,bird,wetlands,omnivore,8,least concern,waterproof feathers,3
elephant,mammal,savannah,herbivore,60,endangered,large ears and trunk,8000
ibis,bird,wetlands,omnivore,16,least concern,"long, curved bill",1
impala,mammal,savannah,herbivore,12,least concern,"long, curved horns",70
kudu,mammal,savannah,herbivore,15,least concern,spiral horns,250
narwhal,mammal,arctic ocean,carnivore,40,near threatened,"long, spiral tusk",
panda,mammal,forests,herbivore,20,vulnerable,black and white coloration,100
polar bear,mammal,arctic,carnivore,25,vulnerable,thick fur and blubber,720
ray,fish,oceans,carnivore,20,"","flat, disc-shaped body",90
```

Важно посмотреть на исходные данные, поскольку обе библиотеки читают их несколько по-разному. Обратите внимание, например, что в данных не указан средний вес нарвала (narwhal), вместо этого строка заканчивается запятой. Кроме того, для ската (ray) не указан статус, а вместо него стоят две двойные кавычки. Попробуйте предположить, как с этими нюансами в исходных данных поступит pandas. Скоро мы узнаем ответ на этот вопрос. А пока посмотрим, что общего между pandas и Polars.

# Сходства pandas и Polars

У библиотек pandas и Polars есть много общего:

- обе представляют собой пакеты на Python для манипулирования структурированными данными[1];
- используют в качестве основной структуры данных датафрейм;
- определяют датафрейм как объединение одного или нескольких объектов Series, в которых должно присутствовать одинаковое количество значений одного типа;

[1] В этой книге мы будем говорить только о Python Polars. Чисто технически Polars написана на языке Rust и располагает привязками к языкам Python, R, JavaScript и Julia.

- поддерживают работу с большим количеством типов данных, включая булевы значения, целочисленные, вещественные, строковые, категориальные, даты, время и временные интервалы;
- обладают обширными API, насчитывающими большое количество функций и методов для работы с датафреймами и объектами Series;
- могут читать и писать в файлы разных форматов и базы данных;
- могут использоваться в Jupyter Notebooks и предлагают HTML-представления;
- начинаются с буквы *p*.

**Что в имени...**

Имя библиотеки pandas на самом деле не имеет ничего общего с пандой как с видом всеядных млекопитающих из семейства медвежьих, а произошло от словосочетания *Panel Data*, описывающего трехмерный датафрейм. Однако уже в версии 0.20.0, вышедшей в мае 2017 года, эта структура данных была упразднена, а имя менять не стали. Сегодня в pandas, как и в Polars, двумя основными структурами данных являются датафрейм и объект Series. Ричи Винк, создатель библиотеки Polars, объяснял выбор имени двумя причинами: во-первых, полярный медведь банально сильнее панды, а во-вторых, Polars оканчивается на `rs`, что является очевидной отсылкой к языку Rust, на котором написана библиотека.

Этого уже достаточно, чтобы водрузиться на льдину и начать дрейфовать от pandas в сторону Polars. В конце концов базовые концепции вам не придется постигать с нуля.

Что ж, пришло время поговорить об отличиях между pandas и Polars. И начнем с самого банального – с внешнего вида.

# Внешний вид pandas и Polars

Как говорится, встречают по одежке. Если вы уже знакомы с pandas, Polars может показаться вам немного странным на первый взгляд. Но давайте посмотрим.

Сначала импортируем обе библиотеки следующим образом:

```
import pandas as pd
import polars as pl
```

Традиционно в качестве алиаса для пакета pandas используется сокращение `pd`, а для Polars – `pl`. Теперь мы можем, наконец, взглянуть на Polars и сравнить синтаксис и вывод с pandas.

## Разница в коде

Давайте прочитаем данные из файла *data/animals.csv* и создадим датафреймы в pandas (с именем `animals_pd`) и в Polars (с именем `animals_pl`):

```
animals_pd = pd.read_csv("data/animals.csv", sep=",", header=0)
animals_pl = pl.read_csv("data/animals.csv", separator=",", has_header=True)

print(f"{type(animals_pd) = }")
print(f"{type(animals_pl) = }")
```

Вывод:

```
type(animals_pd) = <class 'pandas.core.frame.DataFrame'>
type(animals_pl) = <class 'polars.dataframe.frame.DataFrame'>
```

Несмотря на одинаковое имя *`read_csv()`*, эти функции принимают разные аргументы (например, `sep` и `separator`) и возвращают объекты разных типов.

**Все получится!**

При переходе с pandas на Polars вы будете постоянно сталкиваться с такими нюансами и различиями, что может затруднять ваш путь. Но не отчаивайтесь, а просто примите это как должное и верьте нам, когда мы говорим, что все будет в порядке. У вас все получится, мы узнавали!

Теперь посмотрим на разницу в отображении датафреймов и объектов Series в pandas и Polars.

## Разница в выводе

При работе в Jupyter Notebook вы привыкаете к тому, что датафреймы отображаются в красиво оформленном виде с помощью HTML и CSS. На рис. 3.1 и 3.2 показано, как один и тот же датафрейм выглядит в Jupyter Notebook при выводе из pandas и из Polars.

```
[3]: animals_pd
```

[3]:

| | animal | class | habitat | diet | lifespan | status | features | weight |
|---|---|---|---|---|---|---|---|---|
| **0** | dolphin | mammal | oceans/rivers | carnivore | 40 | least concern | high intelligence | 150.0 |
| **1** | duck | bird | wetlands | omnivore | 8 | least concern | waterproof feathers | 3.0 |
| **2** | elephant | mammal | savannah | herbivore | 60 | endangered | large ears and trunk | 8000.0 |
| **3** | ibis | bird | wetlands | omnivore | 16 | least concern | long, curved bill | 1.0 |
| **4** | impala | mammal | savannah | herbivore | 12 | least concern | long, curved horns | 70.0 |
| **5** | kudu | mammal | savannah | herbivore | 15 | least concern | spiral horns | 250.0 |
| **6** | narwhal | mammal | arctic ocean | carnivore | 40 | near threatened | long, spiral tusk | NaN |
| **7** | panda | mammal | forests | herbivore | 20 | vulnerable | black and white coloration | 100.0 |
| **8** | polar bear | mammal | arctic | carnivore | 25 | vulnerable | thick fur and blubber | 720.0 |
| **9** | ray | fish | oceans | carnivore | 20 | NaN | flat, disc-shaped body | 90.0 |

**Рис. 3.1** ❖ Вывод датафрейма pandas в Jupyter Notebook

```
[4]: animals_pl
```

```
[4]: shape: (10, 8)
```

| animal | class | habitat | diet | lifespan | status | features | weight |
|---|---|---|---|---|---|---|---|
| str | str | str | str | i64 | str | str | i64 |
| "dolphin" | "mammal" | "oceans/rivers" | "carnivore" | 40 | "least concern" | "high intelligence" | 150 |
| "duck" | "bird" | "wetlands" | "omnivore" | 8 | "least concern" | "waterproof feathers" | 3 |
| "elephant" | "mammal" | "savannah" | "herbivore" | 60 | "endangered" | "large ears and trunk" | 8000 |
| "ibis" | "bird" | "wetlands" | "omnivore" | 16 | "least concern" | "long, curved bill" | 1 |
| "impala" | "mammal" | "savannah" | "herbivore" | 12 | "least concern" | "long, curved horns" | 70 |
| "kudu" | "mammal" | "savannah" | "herbivore" | 15 | "least concern" | "spiral horns" | 250 |
| "narwhal" | "mammal" | "arctic ocean" | "carnivore" | 40 | "near threatened" | "long, spiral tusk" | null |
| "panda" | "mammal" | "forests" | "herbivore" | 20 | "vulnerable" | "black and white coloration" | 100 |
| "polar bear" | "mammal" | "arctic" | "carnivore" | 25 | "vulnerable" | "thick fur and blubber" | 720 |
| "ray" | "fish" | "oceans" | "carnivore" | 20 | "" | "flat, disc-shaped body" | 90 |

**Рис. 3.2** ❖ Вывод датафрейма Polars в Jupyter Notebook

Обратите внимание на следующие незначительные различия в выводе:

- pandas показывает только таблицу с данными, тогда как Polars также выводит информацию о форме датафрейма, т. е. о количестве строк и столбцов в нем;
- слева от первого столбца с данными pandas выводит технический столбец с индексами, тогда как Polars этого не делает (за неимением таковых);
- pandas отображает только имена столбцов, тогда как Polars также показывает их типы данных;
- в pandas все пропущенные значения отображаются как *NaN*, а в Polars для пропуска в весе нарвала стоит *null*, тогда как отсутствующий статус ската выводится в виде пустой строки (две двойные кавычки);
- в pandas строки показываются как есть, а в Polars заключаются в двойные кавычки.

При работе с Python в терминале вы будете видеть текстовый вывод вместо представлений HTML. Так же мы будем отображать данные и в этой книге. Вот как выглядит вывод в pandas и Polars:

```
animals_pd
```

Вывод:

```
      animal   class        habitat       diet  lifespan           status  \
0     dolphin  mammal  oceans/rivers  carnivore        40    least concern
1        duck    bird       wetlands   omnivore         8    least concern
2    elephant  mammal       savannah  herbivore        60       endangered
3        ibis    bird       wetlands   omnivore        16    least concern
```

```
4      impala  mammal       savannah  herbivore        12    least concern
5        kudu  mammal       savannah  herbivore        15    least concern
6     narwhal  mammal   arctic ocean  carnivore        40  near threatened
7       panda  mammal        forests  herbivore        20       vulnerable
8  polar bear  mammal         arctic  carnivore        25       vulnerable
9         ray    fish         oceans  carnivore        20              NaN

                    features  weight
0           high intelligence   150.0
1         waterproof feathers     3.0
2        large ears and trunk  8000.0
3           long, curved bill     1.0
4          long, curved horns    70.0
5                spiral horns   250.0
6           long, spiral tusk     NaN
7  black and white coloration   100.0
8       thick fur and blubber   720.0
9      flat, disc-shaped body    90.0
```

```
animals_pl
```

Вывод:

```
shape: (10, 8)
+------------+--------+-------------+-----------+----------+-------------+-------------+--------+
| animal     | class  | habitat     | diet      | lifespan | status      | features    | weight |
| ---        | ---    | ---         | ---       | ---      | ---         | ---         | ---    |
| str        | str    | str         | str       | i64      | str         | str         | i64    |
+=============================================================================================+
| dolphin    | mammal | oceans/river| carnivore | 40       | least       | high        | 150    |
|            |        | s           |           |          | concern     | intelligence|        |
| duck       | bird   | wetlands    | omnivore  | 8        | least       | waterproof  | 3      |
|            |        |             |           |          | concern     | feathers    |        |
| elephant   | mammal | savannah    | herbivore | 60       | endangered  | large ears  | 8000   |
|            |        |             |           |          |             | and trunk   |        |
| ibis       | bird   | wetlands    | omnivore  | 16       | least       | long, curved| 1      |
|            |        |             |           |          | concern     | bill        |        |
| impala     | mammal | savannah    | herbivore | 12       | least       | long, curved| 70     |
|            |        |             |           |          | concern     | horns       |        |
| kudu       | mammal | savannah    | herbivore | 15       | least       | spiral horns| 250    |
|            |        |             |           |          | concern     |             |        |
| narwhal    | mammal | arctic ocean| carnivore | 40       | near        | long, spiral| null   |
|            |        |             |           |          | threatened  | tusk        |        |
| panda      | mammal | forests     | herbivore | 20       | vulnerable  | black and   | 100    |
|            |        |             |           |          |             | white       |        |
|            |        |             |           |          |             | coloration  |        |
| polar bear | mammal | arctic      | carnivore | 25       | vulnerable  | thick fur   | 720    |
|            |        |             |           |          |             | and blubber |        |
| ray        | fish   | oceans      | carnivore | 20       |             | flat,       | 90     |
|            |        |             |           |          |             | disc-shaped |        |
|            |        |             |           |          |             | body        |        |
+------------+--------+-------------+-----------+----------+-------------+-------------+--------+
```

Очевидно, что обоим датафреймам трудно поместиться на страницу. При этом разные библиотеки решают эту проблему по-своему. Pandas выводит столько столбцов, сколько может, а затем переходит на новую строку. При этом индекс выводится два раза. Polars, в свою очередь, выводит все столбцы последовательно с переносами на новую строку, а некоторые длинные слова в столбце `features` обрезаются. Также Polars отображает типы данных и форму датафрейма. По умолчанию в этой библиотеке также используются границы, а строки в двойные кавычки при работе в терминале не облачаются.

Теперь сравним вывод на экран объектов Series в обеих библиотеках:

```
animals_pd["animal"]
```

Вывод:

```
0       dolphin
1          duck
2      elephant
3          ibis
4        impala
5          kudu
6       narwhal
7         panda
8    polar bear
9           ray
Name: animal, dtype: object
```

```
animals_pl.get_column("animal")
```

Вывод:

```
shape: (10,)
Series: 'animal' [str]
[
        "dolphin"
        "duck"
        "elephant"
        "ibis"
        "impala"
        "kudu"
        "narwhal"
        "panda"
        "polar bear"
        "ray"
]
```

И снова мы видим, что pandas выводит индекс для каждого элемента объекта Series. Также можно отметить, что pandas присвоил объекту Series тип данных *`object`*.

Обратите внимание, что в pandas не предусмотрено представление объекта Series в виде HTML, тогда как в Polars такое представление присутствует.

В оставшейся части главы и книги мы чаще будем использовать текстовое представление датафреймов. Для комфортного размещения датафреймов `animals_pd` и `animals_pl` на экране можно удалить из них по три столбца. Вот как это делается в pandas и Polars:

```
animals_pd = animals_pd.drop(columns=["habitat", "diet", "features"])
animals_pd
```

Вывод:

```
       animal   class  lifespan           status  weight
0     dolphin  mammal        40    least concern   150.0
1        duck    bird         8    least concern     3.0
2    elephant  mammal        60       endangered  8000.0
3        ibis    bird        16    least concern     1.0
4      impala  mammal        12    least concern    70.0
5        kudu  mammal        15    least concern   250.0
6     narwhal  mammal        40  near threatened     NaN
7       panda  mammal        20       vulnerable   100.0
8  polar bear  mammal        25       vulnerable   720.0
9         ray    fish        20              NaN    90.0
```

```
animals_pl = animals_pl.drop("habitat", "diet", "features")
animals_pl
```

Вывод:

```
shape: (10, 5)
┌────────────┬─────────┬──────────┬──────────────────┬────────┐
│ animal     ┆ class   ┆ lifespan ┆ status           ┆ weight │
│ ---        ┆ ---     ┆ ---      ┆ ---              ┆ ---    │
│ str        ┆ str     ┆ i64      ┆ str              ┆ i64    │
╞════════════╪═════════╪══════════╪══════════════════╪════════╡
│ dolphin    ┆ mammal  ┆ 40       ┆ least concern    ┆ 150    │
│ duck       ┆ bird    ┆ 8        ┆ least concern    ┆ 3      │
│ elephant   ┆ mammal  ┆ 60       ┆ endangered       ┆ 8000   │
│ ibis       ┆ bird    ┆ 16       ┆ least concern    ┆ 1      │
│ impala     ┆ mammal  ┆ 12       ┆ least concern    ┆ 70     │
│ kudu       ┆ mammal  ┆ 15       ┆ least concern    ┆ 250    │
│ narwhal    ┆ mammal  ┆ 40       ┆ near threatened  ┆ null   │
│ panda      ┆ mammal  ┆ 20       ┆ vulnerable       ┆ 100    │
│ polar bear ┆ mammal  ┆ 25       ┆ vulnerable       ┆ 720    │
│ ray        ┆ fish    ┆ 20       ┆                  ┆ 90     │
└────────────┴─────────┴──────────┴──────────────────┴────────┘
```

Мы снова видим границы, форму датафрейма и типы данных в отображении Polars. Pandas же выводит специальный столбец с индексом, о котором мы сейчас и поговорим.

# Концепции, от которых придется отучиться

Моментально забыть о некоторых ключевых концепциях при переходе с pandas на Polars – дело не из легких. Но именно эти концепции определяют, как вы воспринимаете датафреймы и как работаете с предоставленным вам API.

Если вы не сможете отучиться от старых привычек, вам никогда не откроются все грани магии Polars, а разработчики, пришедшие в Polars с нуля, будут писать гораздо более эффективный код, нюансы которого вы даже понять не сможете. В следующих пяти разделах мы перечислим и обсудим ключевые концепции, от которых вам придется отказаться при переходе на Polars, чтобы заставить белого медведя вас слушаться.

## Индексы

Первое, о чем вам следует забыть при переходе из pandas, – это *индексы* (Index). Все датафреймы в pandas обладают индексами, а иногда и *множественными индексами*, или *мультиндексами* (MultiIndex). Зачастую, как в случае с нашим датафреймом `animals_pd`, индекс представляет собой простой объект `RangeIndex`, что видно на примере ниже:

```
animals_pd.index
```

Вывод:

```
RangeIndex(start=0, stop=10, step=1)
```

Некоторые методы, включая методы агрегации данных, способны изменять индексы в датафрейме:

```
animals_agg_pd = animals_pd.groupby(["class", "status"])[["weight"]].mean()
animals_agg_pd
```

Вывод:

```
                          weight
class  status
bird   least concern       2.000000
mammal endangered       8000.000000
       least concern     156.666667
       near threatened          NaN
       vulnerable        410.000000
```

Легко можно убедиться в том, что в результате выполнения этого фрагмента кода был создан множественный индекс на основе столбцов `class` и `status`:

```
animals_agg_pd.index
```

Вывод:

```
MultiIndex([(  'bird',   'least concern'),
            ('mammal',      'endangered'),
            ('mammal',   'least concern'),
            ('mammal', 'near threatened'),
            ('mammal',      'vulnerable')],
           names=['class', 'status'])
```

Не секрет, что далеко не всем разработчикам нравится работать с индексами и мультиндексами в pandas, и они хотели бы от них избавиться. Вы можете передать аргумент `as_index=False` методу `df.groupby()`, чтобы избежать такого поведения, и на практике вы наверняка часто после группировки выполняли инструкцию `df.reset_index()`, чтобы преобразовать созданный индекс в столбцы:

```
animals_agg_pd.reset_index()
```

Вывод:

```
    class           status       weight
0    bird    least concern     2.000000
1  mammal       endangered  8000.000000
2  mammal    least concern   156.666667
3  mammal  near threatened          NaN
4  mammal       vulnerable   410.000000
```

В датафреймах Polars индексы отсутствуют, не говоря уже о множественных индексах, – есть только столбцы. Вот как выглядит подобная агрегация в Polars (о синтаксисе пока не беспокойтесь, скоро все поймете):

```
animals_pl.group_by(["class", "status"]).agg(pl.col("weight").mean())
```

Вывод:

```
shape: (6, 3)
+--------+----------------+------------+
| class  | status         | weight     |
| ---    | ---            | ---        |
| str    | str            | f64        |
+=======================================+
| mammal | least concern  | 156.666667 |
| mammal | near threatened | null       |
| fish   |                | 90.0       |
| bird   | least concern  | 2.0        |
| mammal | vulnerable     | 410.0      |
| mammal | endangered     | 8000.0     |
+--------+----------------+------------+
```

Обратите внимание, что Polars включил в датафрейм строку для класса *fish*, тогда как pandas этого не сделал. Скоро мы вернемся к этому.

Отсутствие индексов в Polars означает, что следующие методы из pandas не имеют аналогов и даже не нуждаются в них: `df.align()`, `df.droplevel()`, `df.reindex()`, `df.rename_axis()`, `df.reset_index()`, `df.set_axis()`, `df.set_index()`, `df.sort_index()`, `df.stack()`, `df.swapaxis()`, `df.swaplevel()` и `df.unstack()`.

# Оси

Следующее, о чем вам придется забыть при переходе из pandas в Polars, – это *оси* (axis). Датафреймы в pandas характеризуются двумя осями: по строкам и по столбцам.

Многие методы для работы с датафреймами в pandas принимают дополнительный параметр `axis` для указания того, к какой оси должна применяться операция. Это такие методы, как `df.drop()`, `df.dropna()`, `df.filter()`, `df.rename()`, `df.shift()`, `df.sort_index()` и `df.sort_values()`. По умолчанию этот аргумент принимает значение `0`, или `rows`. Наверняка вы не раз забывали добавлять этот аргумент при вызове того или иного метода и удивлялись тому, что результат не соответствовал вашим ожиданиям или вовсе не выводился из-за ошибки. К примеру, у вас не получится удалить из датафрейма столбец с именем `weight` с помощью следующей инструкции:

```
animals_pd.drop("weight")
```

Вывод:

```
KeyError: "['weight'] not found in axis"
```

Здесь pandas просто считает, что вы хотите удалить из датафрейма строки, имеющие индекс `weight`. Для удаления столбца вам требуется явным образом передать аргумент `axis=1` или `axis="columns"`. Также с этой целью вы можете воспользоваться аргументом `columns`, как мы делали в начале главы при удалении сразу трех столбцов. Таким образом, для удаления столбца с именем `weight` вы можете выполнить следующую инструкцию:

```
animals_pd.drop("weight", axis=1)
```

Вывод:

```
       animal   class  lifespan           status
0     dolphin  mammal        40    least concern
1        duck    bird         8    least concern
2    elephant  mammal        60       endangered
3        ibis    bird        16    least concern
4      impala  mammal        12    least concern
5        kudu  mammal        15    least concern
6     narwhal  mammal        40  near threatened
```

```
7       panda  mammal        20       vulnerable
8  polar bear  mammal        25       vulnerable
9         ray    fish        20              NaN
```

Конечно, в датафреймах Polars также есть строки и столбцы, но вам больше не нужно заботиться о том, к какой оси вы применяете ту или иную операцию. Методы Polars работают исключительно со столбцами (например, `df.drop()` и `df.rename()`) или со строками (например, `df.filter()`, `df.sort()` и `df.drop_nulls()`).

## Индексирование и срезы

Код на pandas обычно содержит большое количество квадратных скобок, служащих для осуществления индексирования и срезов по строкам и столбцам. В частности, они используются для выбора столбцов, среза в строках, фильтрации строк, создания новых и редактирования существующих столбцов и извлечения значений.

К примеру, для выбора столбцов `animal` и `class` вы обычно пишете следующее:

```
animals_pd[["animal", "class"]]
```

Вывод:

```
       animal   class
0     dolphin  mammal
1        duck    bird
2    elephant  mammal
3        ibis    bird
4      impala  mammal
5        kudu  mammal
6     narwhal  mammal
7       panda  mammal
8  polar bear  mammal
9         ray    fish
```

Чтобы посмотреть животных, находящихся под угрозой, можно воспользоваться следующим фильтром:

```
animals_pd[animals_pd["status"] == "endangered"]
```

Вывод:

```
     animal   class  lifespan      status  weight
2  elephant  mammal        60  endangered  8000.0
```

Для выбора первых трех строк из датафрейма:

```
animals_pd[:3]
```

Вывод:

```
     animal   class  lifespan         status  weight
0   dolphin  mammal        40  least concern   150.0
1      duck    bird         8  least concern     3.0
2  elephant  mammal        60     endangered  8000.0
```

Для обновления существующего столбца:

```
animals_pd["weight"] = animals_pd["weight"] * 1000
```

Кроме того, в pandas квадратным скобкам могут предшествовать атрибуты, как в примерах ниже:

- `df.at[]` – для доступа к одному значению для пары строка/столбец по меткам;
- `df.iat[]` – для доступа к одному значению для пары строка/столбец по целочисленной позиции;
- `df.loc[]` – для доступа к группе строк и столбцов по меткам;
- `df.iloc[]` – для доступа к группе строк и столбцов по целочисленной позиции.

Помимо всего этого многообразия употребления квадратных скобок, в pandas присутствуют такие методы, как `df.xs()`, `df.get()` и `df.filter()`. Как видите, способов для доступа к строкам и столбцам датафреймов в pandas существует великое множество.

В Polars вы ничего такого не увидите. Конечно, квадратные скобки в этой библиотеке тоже используются, но больше с целью совместимости. Главное же правило Polars состоит в том, чтобы *не* использовать скобки. Реализация доступа в Polars обычно осуществляется с помощью специальных методов. Это не только делает ваш код более легким для чтения, но также позволяет воспользоваться оптимизацией при работе в ленивом режиме (подробности – в главе 5).

**Оставьте скобки за скобками**

Как мы уже упоминали, в Polars не приветствуется чрезмерное использование квадратных скобок. Исключение составляют ситуации, когда вам необходимо быстро посмотреть данные в определенных строках и столбцах.

Например, в главе 1 мы демонстрировали пример выбора некоторого количества столбцов из широкого датафрейма:

```
print(trips[:, :4])
print(trips[:, 4:8])
print(trips[:, 8:])
```

Иных причин использования квадратных скобок в коде на Polars не существует.

Еще одним следствием присутствия индексов в pandas является то, что столбцы автоматически упорядочиваются по существующему в датафрейме индексу. Рассмотрим следующий пример со столбцом:

```
animals_pd["weight"]
```

Вывод:

```
0     150.0
1       3.0
2    8000.0
3       1.0
4      70.0
5     250.0
6       NaN
7     100.0
8     720.0
9      90.0
Name: weight, dtype: float64
```

При необходимости отсортировать значения в этом столбце и вернуть его в исходный датафрейм вы могли бы сделать следующее:

```
animals_pd["weight"] = animals_pd["weight"].sort_values()
```

Однако поскольку новые значения будут отсортированы в соответствии с существующим индексом, исходный датафрейм не изменится, несмотря на затраченное время на сортировку:

```
animals_pd["weight"]
```

Вывод:

```
0     150.0
1       3.0
2    8000.0
3       1.0
4      70.0
5     250.0
6       NaN
7     100.0
8     720.0
9      90.0
Name: weight, dtype: float64
```

Это необычный побочный эффект от использования индексов.

# Жадность

В pandas все операции выполняются в жадной манере непосредственно в момент их вызова и зачастую прямо на месте. *Жадная парадигма* (eager) как раз предполагает безотлагательное осуществление операций, а выполнение *на месте* (in place) подразумевает, что метод не возвращает новый датафрейм, а изменяет существующий. Ранее вы уже видели подобный пример, когда мы обновляли столбец `weight`.

В Polars ни одна операция не выполняется на месте по умолчанию[1].

Помимо жадного режима, библиотека Polars также способна работать в ленивой парадигме. В этом случае вы начинаете инструкцию с определения ленивого датафрейма (LazyFrame вместо DataFrame), перечисляете операции, которые хотите выполнить, и завершаете инструкцию вызовом метода `lf.collect()`. В результате Polars оптимизирует запрос и только после этого отправляет его на выполнение.

Ниже приведен пример ленивого запроса, в котором мы читаем данные из файла *data/animals.csv*, рассчитываем средний вес для каждого класса животных и оставляем в выводе только млекопитающих:

```
lazy_query = (
    pl.scan_csv("data/animals.csv")
    .group_by("class")
    .agg(pl.col("weight").mean())
    .filter(pl.col("class") == "mammal")
)
```

Обратите внимание, что функция *`pl.scan_csv()`* не выполняет чтение файла непосредственно в момент ее появления в запросе.

Понять, что на самом деле происходит в этом запросе, можно, если взглянуть на *план запроса* (query plan). Этот план содержит вычислительные шаги, которые будут запущены в момент выполнения запроса. План запроса удобно читать снизу вверх. В главе 18 мы будем подробно говорить о планах запросов, поскольку эта тема требует хорошего понимания внутреннего устройства Polars. А сейчас мы можем приоткрыть завесу и взглянуть на святая святых библиотеки Polars, а именно ее оптимизатор.

На рис. 3.3 показан неоптимизированный (наивный) план нашего запроса.

```
lazy_query.show_graph(optimized=False)
```

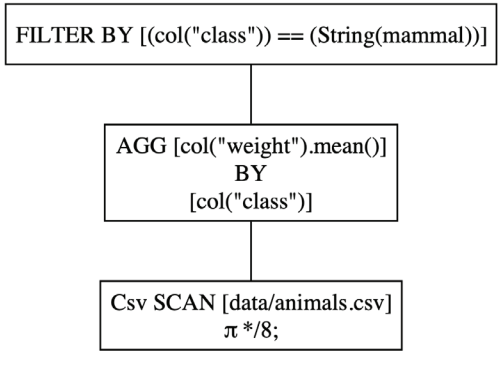



**Рис. 3.3** ❖ Наивный план запроса

[1] В Polars всего пара методов может выполняться на месте, а именно методы `hstack()` и `shrink_to_fit()` при указании аргумента `in_place=True`.

На рис. 3.4 приведен оптимизированный план выполнения запроса.

```
lazy_query.show_graph()
```

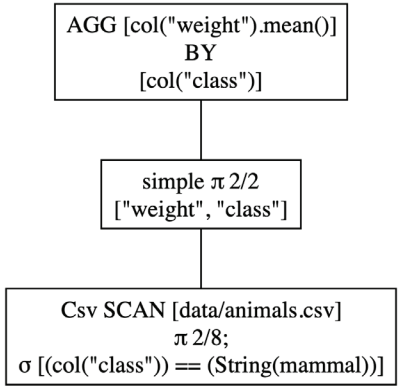



**Рис. 3.4** ❖ Оптимизированный план запроса

Как видите, Polars распознала, что эффективнее будет выполнить фильтрацию *перед* агрегацией. Шаг с чтением файла также содержит операцию фильтрации. Это позволяет сэкономить время двумя способами. Во-первых, мы читаем из файла только нужные нам данные. Во-вторых, последующие шаги обрабатывают только то, что вам действительно нужно. С помощью метода *`collect()`* можно материализовать датафрейм:

```
lazy_query.collect()
```

Вывод:

```
shape: (1, 2)
+--------+-------------+
| class  | weight      |
| ---    | ---         |
| str    | f64         |
+======================+
| mammal | 1548.333333 |
+--------+-------------+
```

Подробно о ленивых вычислениях мы будем говорить в главе 5. Конечно, использование ленивой парадигмы с таким маленьким набором данных не столь оправдано, но смысл должен быть понятен. Даже если вы не будете использовать ленивый API, ваш код на Polars будет выглядеть похожим образом – с большим количеством методов и (мы надеемся) практически без скобок.

## Вседозволенность

Последнее, о чем вам придется забыть при переходе на Polars, – это о так называемой вседозволенности pandas. Иными словами, привыкайте к строгим правилам Polars.

Это не какая-то единая концепция, а некий набор характеристик библиотеки. Вот неполный список, описывающий вседозволенность pandas:

- pandas позволяет использовать строковые типы при работе с датой и временем. В Polars вы обязаны использовать тип данных `datetime` из Python или типы `Date` или `Datetime` из Polars. Подробнее об этих типах данных мы поговорим в главе 12;
- в pandas вы можете использовать в качестве имен столбцов не только строковые значения. В Polars допустимы только строки;
- в pandas существует масса способов отображения и обработки пропущенных значений, например `None`, `NaN`, `NA`, `<NA>`, `NaT` и `np.nan`, в зависимости от типов данных. В Polars делается различие между пропущенными значениями (*`null`*) и некорректными числами (*`NaN`*, или «Not a Number»), и они применяются одинаково вне зависимости от типов данных;
- в pandas при наличии в целочисленном столбце пропущенных значений тип этого столбца изменяется на вещественный (этого не происходит при использовании новых типов данных Arrow). Причина в том, что целочисленные столбцы не могут содержать пропуски. В Polars таких проблем нет, и пропущенные значения могут присутствовать в столбцах любых типов;
- pandas молча исключает из набора данных строки с пропущенными значениями при выполнении агрегаций (см. предыдущий пример с исчезнувшей строкой для класса *fish*). Polars сохраняет все группы;
- в pandas колонки могут приобретать тип данных object, даже если на самом деле в них хранятся строки или вещественные числа. Polars никогда не присваивает столбцам обобщенный тип данных;
- в pandas вы можете сравнивать объекты Series булева типа с целочисленными значениями. В Polars булевы значения можно сравнивать только с булевыми;
- в pandas у вас может быть несколько столбцов в датафрейме с одинаковыми именами. В Polars имена столбцов должны быть уникальными;
- в pandas методы и свойства, такие как `df.loc[]`, возвращают объект Series, а не датафрейм, если передается только одно значение (исключение составляет ситуация, когда присутствует несколько столбцов с одним именем, тогда возвращается датафрейм). В Polars в таких случаях возвращается датафрейм с одной строкой;
- в pandas выражение `df.sum(axis=0)` вернет объект Series. В Polars аналогичная операция вернет датафрейм с одной строкой.

Как вы вскоре узнаете, такая строгость Polars способна принести неплохие дивиденды в виде быстрого отклика о том, что запрос может завершиться ошибкой, гарантии того, что результаты будут корректными, и применения оптимизаций. Короче говоря, строгость Polars – это не баг, а фича.

# Синтаксис, который придется забыть

Отучившись от этих концепций, вам необходимо будет также отказаться и от некоторых мелких привычек, приобретенных за время использования pandas. Это, в частности, касается названия функций, методов и аргументов. Процесс переучивания может затянуться, но документация поможет его облегчить.

**Сложности перевода**

Вы можете поддаться соблазну и начать переводить свой код, написанный в pandas, на язык Polars построчно. Спешим вас предостеречь, что в этом случае вы можете столкнуться с теми же сложностями, которые поджидают переводчиков, пытающихся перекладывать нидерландский язык на английский по словам, – никакого дерна вы на дамбу не положите[1]. Иначе говоря, обычно это вам не поможет. Чаще всего в таком случае вы получите код, не характерный для Polars, который может работать крайне неэффективно.

## Сравнение некоторых операций

Давайте посмотрим, как выполняются распространенные операции для работы с датафреймами в pandas и Polars. Конечно, это не полный список операций, но он поможет вам понять, насколько по-разному эти библиотеки подходят к обработке данных. Мы также покажем разницу в выводах.

### *Удаление дубликатов*

Допустим, нам нужно удалить некоторые строки в датафрейме, чтобы в столбце `class` не осталось дубликатов. В pandas мы бы это сделали так:

```
animals_pd.drop_duplicates(subset="class")
```

Вывод:

```
    animal   class  lifespan         status  weight
0  dolphin  mammal        40  least concern   150.0
```

[1] Фраза «не класть дерн на дамбу» (puts no sods on the dike – англ.) является дословным переводом голландского выражения «Dat zet geen zoden aan de dijk», означающего, что предложенный вариант не поможет решить проблему.

```
1     duck    bird         8  least concern     3.0
9      ray    fish        20            NaN    90.0
```

В Polars мы напишем так:

```
animals_pl.unique(subset="class")
```

Вывод:

```
shape: (3, 5)
+---------+--------+----------+--------------+--------+
| animal  | class  | lifespan | status       | weight |
| ---     | ---    | ---      | ---          | ---    |
| str     | str    | i64      | str          | i64    |
+=================================================+
| ray     | fish   | 20       |              | 90     |
| duck    | bird   | 8        | least concern | 3      |
| dolphin | mammal | 40       | least concern | 150    |
+---------+--------+----------+--------------+--------+
```

Обратите внимание на разницу в названии методов: в pandas метод называется *drop_duplicates()*, а в Polars – *unique()*. Также отметим, что в Polars не сохранился исходный порядок строк (скат (ray) оказался выше утки (duck)). Причина в том, что Polars выполняет запросы параллельно, что может приводить к изменению порядка строк из-за разницы во времени их обработки. Такое поведение можно изменить с помощью аргумента *maintain_order*=True, но в этом случае вам придется пожертвовать скоростью выполнения.

## *Удаление пропущенных значений*

Теперь давайте удалим строки, в которых есть пропущенные значения в столбце weight. На pandas мы бы сделали это так:

```
animals_pd.dropna(subset="weight")
```

Вывод:

```
       animal   class  lifespan         status  weight
0     dolphin  mammal        40  least concern   150.0
1        duck    bird         8  least concern     3.0
2    elephant  mammal        60     endangered  8000.0
3        ibis    bird        16  least concern     1.0
4      impala  mammal        12  least concern    70.0
5        kudu  mammal        15  least concern   250.0
7       panda  mammal        20     vulnerable   100.0
8  polar bear  mammal        25     vulnerable   720.0
9         ray    fish        20            NaN    90.0
```

А на Polars эта операция выглядела бы так:

```
animals_pl.drop_nulls(subset="weight")
```

Вывод:

```
shape: (9, 5)
+------------+--------+----------+--------------+--------+
| animal     | class  | lifespan | status       | weight |
| ---        | ---    | ---      | ---          | ---    |
| str        | str    | i64      | str          | i64    |
+=============================================================+
| dolphin    | mammal | 40       | least concern | 150    |
| duck       | bird   | 8        | least concern | 3      |
| elephant   | mammal | 60       | endangered    | 8000   |
| ibis       | bird   | 16       | least concern | 1      |
| impala     | mammal | 12       | least concern | 70     |
| kudu       | mammal | 15       | least concern | 250    |
| panda      | mammal | 20       | vulnerable    | 100    |
| polar bear | mammal | 25       | vulnerable    | 720    |
| ray        | fish   | 20       |               | 90     |
+------------+--------+----------+--------------+--------+
```

Здесь также отличаются имена методов: *dropna()* в pandas против *drop_nulls()* в Polars.

## Сортировка строк

Давайте упорядочим данные в наборе по столбцу `weight` по убыванию (самые тяжелые животные должны возглавить таблицу). В pandas:

```
animals_pd.sort_values("weight", ascending=False)
```

Вывод:

```
      animal   class  lifespan           status  weight
2   elephant  mammal        60       endangered  8000.0
8  polar bear  mammal        25       vulnerable   720.0
5        kudu  mammal        15    least concern   250.0
0     dolphin  mammal        40    least concern   150.0
7       panda  mammal        20       vulnerable   100.0
9         ray    fish        20              NaN    90.0
4      impala  mammal        12    least concern    70.0
1        duck    bird         8    least concern     3.0
3        ibis    bird        16    least concern     1.0
6     narwhal  mammal        40  near threatened     NaN
```

В Polars:

```
animals_pl.sort("weight", descending=True)
```

Вывод:

```
shape: (10, 5)
+------------+--------+----------+----------------+--------+
| animal     | class  | lifespan | status         | weight |
| ---        | ---    | ---      | ---            | ---    |
| str        | str    | i64      | str            | i64    |
```

```
+=========================================================+
| narwhal    | mammal | 40       | near threatened | null   |
| elephant   | mammal | 60       | endangered      | 8000   |
| polar bear | mammal | 25       | vulnerable      | 720    |
| kudu       | mammal | 15       | least concern   | 250    |
| dolphin    | mammal | 40       | least concern   | 150    |
| panda      | mammal | 20       | vulnerable      | 100    |
| ray        | fish   | 20       |                 | 90     |
| impala     | mammal | 12       | least concern   | 70     |
| duck       | bird   | 8        | least concern   | 3      |
| ibis       | bird   | 16       | least concern   | 1      |
+------------+--------+----------+-----------------+--------+
```

Здесь разница состоит как в названии методов (*sort_values()* против *sort()*), так и в имени и семантике аргумента: (`ascending=False` против `descending=True`). Кроме того, pandas располагает пропущенные значения в сортируемой колонке в конце списка, а Polars – в начале. Такое поведение можно изменить с помощью аргумента `nulls_last=True`.

## *Приведение типов существующих столбцов*

Теперь изменим тип данных столбца `lifespan` с целочисленного на вещественный. В pandas мы бы сделали это так:

```
animals_pd.assign(lifespan=animals_pd["lifespan"].astype(float))
```

Вывод:

```
       animal   class  lifespan           status  weight
0     dolphin  mammal      40.0    least concern   150.0
1        duck    bird       8.0    least concern     3.0
2    elephant  mammal      60.0       endangered  8000.0
3        ibis    bird      16.0    least concern     1.0
4      impala  mammal      12.0    least concern    70.0
5        kudu  mammal      15.0    least concern   250.0
6     narwhal  mammal      40.0  near threatened     NaN
7       panda  mammal      20.0       vulnerable   100.0
8  polar bear  mammal      25.0       vulnerable   720.0
9         ray    fish      20.0              NaN    90.0
```

В большинстве случаев разработчики в pandas предпочитают изменять столбцы напрямую с помощью символа присваивания (`=`), а не метода *assign()*.

В Polars:

```
animals_pl.with_columns(pl.col("lifespan").cast(pl.Float64))
```

Вывод:

```
shape: (10, 5)
+------------+--------+----------+-----------------+--------+
| animal     | class  | lifespan | status          | weight |
| ---        | ---    | ---      | ---             | ---    |
```

```
| str        | str    | f64      | str             | i64    |
+=============================================================+
| dolphin    | mammal | 40.0     | least concern   | 150    |
| duck       | bird   | 8.0      | least concern   | 3      |
| elephant   | mammal | 60.0     | endangered      | 8000   |
| ibis       | bird   | 16.0     | least concern   | 1      |
| impala     | mammal | 12.0     | least concern   | 70     |
| kudu       | mammal | 15.0     | least concern   | 250    |
| narwhal    | mammal | 40.0     | near threatened | null   |
| panda      | mammal | 20.0     | vulnerable      | 100    |
| polar bear | mammal | 25.0     | vulnerable      | 720    |
| ray        | fish   | 20.0     |                 | 90     |
+------------+--------+----------+-----------------+--------+
```

В Polars используется функция `pl.col()` для создания выражения. Выражения – это отдельная большая тема в Polars, и подробно о ней мы будем говорить в третьей части книги.

## *Агрегация данных*

Давайте сгруппируем таблицу по столбцам `class` и `status` и рассчитаем для каждой группы среднее значение по столбцу `weight`. Мы уже делали что-то подобное, но не лишним будет повторить. В pandas эта операция будет выглядеть так:

```
animals_pd.groupby(["class", "status"])[["weight"]].mean()
```

Вывод:

```
                          weight
class  status
bird   least concern       2.000000
mammal endangered       8000.000000
       least concern     156.666667
       near threatened          NaN
       vulnerable        410.000000
```

А в Polars:

```
animals_pl.group_by("class", "status").agg(pl.col("weight").mean())
```

Вывод:

```
shape: (6, 3)
+--------+----------------+------------+
| class  | status         | weight     |
| ---    | ---            | ---        |
| str    | str            | f64        |
+=======================================+
| mammal | endangered     | 8000.0     |
| mammal | vulnerable     | 410.0      |
| mammal | least concern  | 156.666667 |
```

```
| mammal | near threatened | null       |
| fish   |                 | 90.0       |
| bird   | least concern   | 2.0        |
+--------+-----------------+------------+
```

Разница в названиях методов здесь небольшая, но важная (*groupby()* против *group_by()*). Также pandas создает множественный индекс на основе столбцов `class` и `status`, тогда как в Polars они остаются в виде отдельных столбцов. Кроме того, Polars использует выражение внутри метода *agg()*, что позволяет строить собственные сложные агрегации. В то же время pandas ограничен списком предопределенных агрегаций (`min`, `max`, `first`, `mean`, `std` и несколько других). Подробнее об агрегациях мы будем говорить в главе 13.

# Из Polars в pandas и обратно

К счастью, мы можем легко преобразовывать датафреймы pandas в Polars и наоборот. Самый простой и очевидный способ опробовать Polars состоит в преобразовании какого-нибудь своего фрагмента кода из pandas в Polars и его проверке. Перекинуть датафрейм из pandas в Polars можно следующим образом:

```
animals_pl = pl.DataFrame(animals_pd)
animals_pl
```

Вывод:

```
shape: (10, 5)
+------------+--------+----------+-----------------+--------+
| animal     | class  | lifespan | status          | weight |
| ---        | ---    | ---      | ---             | ---    |
| str        | str    | i64      | str             | f64    |
+=============================================================+
| dolphin    | mammal | 40       | least concern   | 150.0  |
| duck       | bird   | 8        | least concern   | 3.0    |
| elephant   | mammal | 60       | endangered      | 8000.0 |
| ibis       | bird   | 16       | least concern   | 1.0    |
| impala     | mammal | 12       | least concern   | 70.0   |
| kudu       | mammal | 15       | least concern   | 250.0  |
| narwhal    | mammal | 40       | near threatened | null   |
| panda      | mammal | 20       | vulnerable      | 100.0  |
| polar bear | mammal | 25       | vulnerable      | 720.0  |
| ray        | fish   | 20       | null            | 90.0   |
+------------+--------+----------+-----------------+--------+
```

Это по крайней мере позволит вам начать работать с Polars на практике. Когда вам захочется вернуться обратно, вы можете воспользоваться методом *to_pandas()*:

```
there_and_back_again_df = animals_pl.to_pandas()
there_and_back_again_df
```

Вывод:

```
       animal   class  lifespan           status  weight
0     dolphin  mammal        40    least concern   150.0
1        duck    bird         8    least concern     3.0
2    elephant  mammal        60       endangered  8000.0
3        ibis    bird        16    least concern     1.0
4      impala  mammal        12    least concern    70.0
5        kudu  mammal        15    least concern   250.0
6     narwhal  mammal        40  near threatened     NaN
7       panda  mammal        20       vulnerable   100.0
8  polar bear  mammal        25       vulnerable   720.0
9         ray    fish        20             None    90.0
```

# Заключение

В этой главе мы посмотрели на сходства и различия pandas и Polars. Ниже приведены ключевые выводы:

- без pandas, скорее всего, не было бы и Polars;
- у pandas и Polars есть не так много общего;
- в то же время датафреймы Polars довольно похожи на датафреймы pandas, за несколькими важными исключениями;
- при переходе на Polars вам придется забыть о некоторых ключевых концепциях, которые вы использовали в pandas. В первую очередь это касается индексов, осей, индексирования и срезов, жадной парадигмы выполнения и своеобразной вседозволенности pandas;
- большинство основных методов в pandas и Polars называются по-разному и принимают разные аргументы, что может доставлять некоторые неудобства;
- некоторые схожие операции предполагают разный вывод;
- существуют простые способы преобразования датафреймов pandas в Polars и обратно.

Мы понимаем, что было довольно жестоко просить вас напрочь забыть обо всем, что вы узнали, работая с pandas. Но в следующих главах книги мы возместим вам моральный ущерб и попробуем в простой и непринужденной манере познакомить вас со всеми прелестями Polars.

# ЧАСТЬ II

# Форматы и структуры

# Глава 4

# Структуры и форматы данных

Теперь, когда вы познакомились с Polars в общих чертах и даже посмотрели несколько примеров, пришло время узнать, как на самом деле работает эта библиотека и в чем ее сильные стороны. В Polars для хранения данных используется спецификация Arrow, предлагающая множество типов данных.

В этой главе вы познакомитесь:

- со структурами, использующимися в Polars для хранения данных;
- с доступными в Polars типами данных;
- с типами данных, работа с которыми является не столь очевидной.

Все инструкции по загрузке нужных файлов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

## Series, DataFrame и LazyFrame

Polars хранит все свои данные в объектах Series и датафреймах.

Объект *Series* представляет собой одномерную структуру данных, способную хранить последовательность значений. Все значения в одном объекте Series должны обладать одним типом данных. Объекты Series могут использоваться как сами по себе, так и в составе датафрейма в виде столбцов.

Пример Series:

```
sales_series = pl.Series("sales", [150.00, 300.00, 250.00])
sales_series
```

Вывод:

```
shape: (3,)
Series: 'sales' [f64]
[
        150.0
```

```
        300.0
        250.0
]
```

*Датафрейм* (DataFrame) – это двумерная структура данных, в которой значения организованы в виде таблицы, со строками и столбцами. В Polars датафрейм хранится как последовательность объектов Series одной длины. С внутренним устройством датафреймов и Series мы познакомимся в главе 18.

Пример датафрейма на основе созданного ранее объекта Series:

```
sales_df = pl.DataFrame(
    {
        "sales": sales_series,
        "customer_id": [24, 25, 26],
    }
)
sales_df
```

Вывод:

```
shape: (3, 2)
+-------+-------------+
| sales | customer_id |
| ---   | ---         |
| f64   | i64         |
+=====================+
| 150.0 | 24          |
| 300.0 | 25          |
| 250.0 | 26          |
+-------+-------------+
```

*Ленивый датафрейм* (*LazyFrame*) – это датафрейм без данных[1]. Тогда как датафрейм хранит данные напрямую в памяти, ленивый датафрейм содержит только инструкции для чтения и обработки данных. При этом ни одна операция чтения или преобразования данных не выполняется в ленивом датафрейме немедленно. Вместо этого все операции откладываются до момента, когда они действительно понадобятся. В этом и состоит парадигма ленивых вычислений. До момента фактического выполнения ленивый датафрейм пребывает в состоянии чернового проекта для будущего датафрейма – по сути, в виде графа, состоящего из вычислительных шагов. Этот граф позволяет оптимизатору построить более эффективный план выполнения запроса.

На рис. 4.1 представлен пример графа для ленивого датафрейма.

---

[1] Ленивый датафрейм может содержать данные, когда он возникает из обычного датафрейма путем вызова метода `df.lazy()`. В этом случае исходный датафрейм хранится в ленивом. Помимо этого, в ленивом датафрейме также могут содержаться метаданные с информацией о данных в источнике, чтобы оптимизатор мог творить свою магию.

```
lazy_df = pl.scan_csv("data/fruit.csv").with_columns(
    is_heavy=pl.col("weight") > 200
)

lazy_df.show_graph()
```

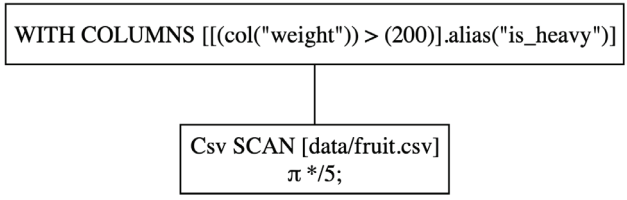



**Рис. 4.1** ❖ Граф выполнения для ленивого датафрейма

При вызове метода `lf.collect()` этот граф с инструкциями будет выполнен, в результате чего ленивый датафрейм материализуется. Подробнее о жадном и ленивом API мы будем говорить в главе 5.

# Типы данных

Для эффективного хранения данных в Polars используется спецификация работы с памятью *Apache Arrow*. В главе 18 мы более детально будем говорить о технологии Arrow и о том, как она работает. Если говорить коротко, Arrow представляет собой колоночный формат хранения плоских и иерархических данных в памяти, оптимизированный для эффективного выполнения аналитических операций с использованием современных центральных и графических процессоров. Таким образом, в Polars данные хранятся в виде, идеально подходящем для их быстрого анализа и обработки.

В Polars используются типы данных, показанные в табл. 4.1. Большинство из них базируются на спецификации Arrow[1]. Некоторые типы присутствуют в таблице по нескольку раз для разных размеров. Это позволяет значительно экономить фактическую память при хранении данных.

[1] Иногда Polars отклоняется от спецификации, принятой в Arrow. К примеру, в Polars реализован собственный тип данных `String` для лучшей эффективности. К тому же в Arrow отсутствуют типы данных `Object` и `Unknown`. С полным списком типов данных спецификации Arrow можно ознакомиться по адресу *https://arrow.apache.org/docs/python/api/datatypes.html*.

***Таблица 4.1. Типы данных, используемые в Polars***

| Группа | Тип данных | Описание | Диапазон |
|---|---|---|---|
| Numeric | `DataType` | Базовый класс для всех типов данных в Polars | |
| | `Decimal` | Десятичный 128-битный тип с опциональной точностью и неотрицательным масштабом | Может использоваться для явного представления 38 значимых цифр |
| | `Float32` | 32-битный тип с плавающей запятой | от –3.4e+38 до 3.4e+38 |
| | `Float64` | 64-битный тип с плавающей запятой | от –1.7e+308 до 1.7e+308 |
| | `Int8` | 8-битный целочисленный тип со знаком | от –128 до 128 |
| | `Int16` | 16-битный целочисленный тип со знаком | от –32 768 до 32 767 |
| | `Int32` | 32-битный целочисленный тип со знаком | от –2 147 483 648 до 2 147 483 647 |
| | `Int64` | 64-битный целочисленный тип со знаком | от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 |
| | `UInt8` | 8-битный целочисленный тип без знака | от 0 до 255 |
| | `UInt16` | 16-битный целочисленный тип без знака | от 0 до 65 535 |
| | `UInt32` | 32-битный целочисленный тип без знака | от 0 до 4 294 967 295 |
| | `UInt64` | 64-битный целочисленный тип без знака | от 0 до 1.8446744e+19 |
| Temporal | `Date` | Календарный тип для даты. Внутренне использует тип данных date32 из Arrow для представления количества дней, прошедших с начала эпохи Unix (1 января 1970 года) | от –5877641-06-24 до 5879610-09-09 |
| | `Datetime` | Календарный тип для даты и времени. Точная временная метка на основе начала эпохи Unix с использованием типа int64. По умолчанию представлен с точностью до миллисекунд | |
| | `Duration` | Тип данных для хранения временных промежутков | |
| | `Time` | Тип данных для хранения времени | |
| Nested | `Array` | Тип данных List с фиксированной длиной | |
| | `List` | Тип данных List с переменной длиной | |
| | `Struct` | Тип Struct | |

***Таблица 4.1*** *(окончание)*

| Группа | Тип данных | Описание | Диапазон |
|---|---|---|---|
| String | `String` | Строковый тип произвольной длины с использованием кодировки UTF-8 | |
| | `Categorical` | Категориально закодированный набор строк. Позволяет более эффективно использовать память, если в объекте Series содержится не так много уникальных значений | |
| | `Enum` | Фиксированный категориально закодированный набор строк. Категории должны быть известны и определены заранее | |
| Other | `Boolean` | Булев тип с использованием 1 бита | True или False |
| | `Binary` | Двоичный тип с произвольным количеством байтов | |
| | `Null` | Тип данных, представляющий значения Null/None | |
| | `Object` | Тип-обертка для произвольных объектов Python | |
| | `Unknown`[1] | Тип, представляющий значения типа данных, которые не могут быть определены статически | |

**Хитрый тип Object**

Иногда возникает необходимость добавить в датафрейм некие произвольные объекты Python. К примеру, вам может понадобиться хранить в столбце целые модели машинного обучения. В этом случае вы можете воспользоваться специальным типом данных *`Object`*.

Недостатком этого подхода является то, что такие данные не могут быть обработаны с помощью обычных функций. Кроме того, к ним не могут быть применены никакие методы оптимизации, поскольку Polars не использует Python для просмотра содержимого объектов. В результате значения с типом данных `Object` становятся обычными «пассажирами» в датафрейме, которые можно передавать, к примеру, в операции объединения, но которые не будут принимать участия в процессе оптимизации.

Обычно, если возможно представить данные с использованием других типов, применять тип `Object` не рекомендуется, но бывают и редкие исключения.

# Вложенные типы данных

В Polars присутствуют три вложенных типа данных: `Array`, `List` и `Struct`. С помощью этих типов вы можете эффективно хранить сложные структуры дан-

[1] В документации упоминается тип данных `Unknown`, который используется внутренне и не предназначен для применения в коде.

ных в датафреймах. Тип *Array* представляет собой коллекцию фиксированной длины, состоящую из значений одного типа. Обычно используется для компактного хранения данных и предсказуемого индексирования. Тип данных *List* является более гибким в сравнении с Array, поскольку позволяет хранить списки разной длины в каждой строке. Наконец, тип *Struct* дает возможность инкапсулировать несколько именованных полей в одну сущность.

При использовании внутри объекта Series все элементы типа Array должны обладать одинаковым размером. При этом сама размерность может быть любой. К примеру, для хранения пикселей цветного изображения размером 640 на 480 вы могли бы использовать три измерения. Определить внутренний тип данных и форму объектов Array можно следующим образом:

```
coordinates = pl.DataFrame(
    [
        pl.Series("point_2d", [[1, 3], [2, 5]]),
        pl.Series("point_3d", [[1, 7, 3], [8, 1, 0]]),
    ],
    schema={
        "point_2d": pl.Array(shape=2, inner=pl.Int64),
        "point_3d": pl.Array(shape=3, inner=pl.Int64),
    },
)

coordinates
```

Вывод:

```
shape: (2, 2)
┌─────────────────┬─────────────────┐
│ point_2d        ┆ point_3d        │
│ ---             ┆ ---             │
│ array[i64, 2]   ┆ array[i64, 3]   │
╞═════════════════╪═════════════════╡
│ [1, 3]          ┆ [1, 7, 3]       │
│ [2, 5]          ┆ [8, 1, 0]       │
└─────────────────┴─────────────────┘
```

Тип данных List похож на тип Array в том, что также позволяет хранить коллекции значений одного типа данных. Однако, в отличие от Array, тип данных List не накладывает ограничения на количество элементов в каждой строке. Обратите внимание, что этот тип отличается от стандартных списков в Python, которые могут содержать элементы разных типов. Списки Python можно хранить в объекте Series, определив для него тип данных Object. Единственным аргументом типа List является тип хранящихся данных.

Ниже показан пример создания датафрейма с двумя столбцами типа List. Поскольку здесь мы не определяем схему явно, как делали в предыдущем примере, внутренние типы будут выведены из самих данных:

```
weather_readings = pl.DataFrame(
    {
        "temperature": [[72.5, 75.0, 77.3], [68.0, 70.2]],
```

```
        "wind_speed": [[15, 20], [10, 12, 14, 16]],
    }
)

weather_readings
```

Вывод:

```
shape: (2, 2)
┌─────────────────┬──────────────┐
│ temperature     ┆ wind_speed   │
│ ---             ┆ ---          │
│ list[f64]       ┆ list[i64]    │
╞═════════════════╪══════════════╡
│ [72.5, 75.0, 77.3] ┆ [15, 20]     │
│ [68.0, 70.2]    ┆ [10, 12, … 16] │
└─────────────────┴──────────────┘
```

Теперь посмотрим на тип данных `Struct`. Часто этот тип данных используется, чтобы работать с несколькими объектами Series одновременно. Ниже показано, как можно создать столбцы с типом `Struct` на основе словарей в Python:

```
rating_series = pl.Series(
    "ratings",
    [
        {"Movie": "Cars", "Theatre": "NE", "Avg_Rating": 4.5},
        {"Movie": "Toy Story", "Theatre": "ME", "Avg_Rating": 4.9},
     ],
)

rating_series
```

Вывод:

```
shape: (2,)
Series: 'ratings' [struct[3]]
[
        {"Cars","NE",4.5}
        {"Toy Story","ME",4.9}
]
```

Более подробно о работе с типами данных `List`, `Array` и `Struct` мы поговорим в главе 12.

# Пропущенные значения

В Polars *пропущенные значения* всегда представляются как *null*. Это справедливо для всех типов данных, включая числовые[1]. Информация о пропущенных значениях хранится в метаданных объекта Series.

[1] Исключение составляет сам тип `Null`, который не может представлять пропущенные значения.

Также сведения о пропущенных значениях хранятся в *проверочной битовой карте* (validity bitmap), в которой единички соответствуют присутствующим значениям, а нолики – пропущенным. Это позволяет вам легко узнать, сколько пропусков есть в объекте Series, с помощью таких методов, как *null_count()* и *Expr.is_null()*. Для демонстрации создадим датафрейм, содержащий пропущенные значения:

```
missing_df = pl.DataFrame(
    {
        "value": [None, 2, 3, 4, None, None, 7, 8, 9, None],
    },
)

missing_df
```

Вывод:

```
shape: (10, 1)
┌───────┐
│ value │
│ ---   │
│ i64   │
╞═══════╡
│ null  │
│ 2     │
│ 3     │
│ 4     │
│ null  │
│ null  │
│ 7     │
│ 8     │
│ 9     │
│ null  │
└───────┘
```

Заполнить пропуски в столбце можно с помощью метода *Expr.fill_null()* с использованием четырех разных стратегий:

- заполнение одним значением;
- заполнение в соответствии с выбранной стратегией;
- заполнение с использованием выражения;
- заполнение с использованием интерполяции.

**Не число, но и не пропуск**

Значения *NaN* (означает «не число», «not a number») не считаются в Polars пропущенными. Эти значения могут присутствовать в столбцах с вещественными числами и говорят о том, что результат операции не является числовым. Соответственно, значения *NaN* не воспринимаются как *null* такими методами, как *null_count()* или *Expr.fill_null()*. Для определения таких значений можно воспользоваться методами *Expr.is_nan()* и *Expr.fill_nan()*.

В следующем примере мы покажем, как можно заполнить пропуски фиксированным значением:

```
missing_df.with_columns(filled_with_single=pl.col("value").fill_null(-1))
```

Вывод:

```
shape: (10, 2)
┌───────┬────────────────────┐
│ value ┆ filled_with_single │
│ ---   ┆ ---                │
│ i64   ┆ i64                │
╞═══════╪════════════════════╡
│ null  ┆ -1                 │
│ 2     ┆ 2                  │
│ 3     ┆ 3                  │
│ 4     ┆ 4                  │
│ null  ┆ -1                 │
│ null  ┆ -1                 │
│ 7     ┆ 7                  │
│ 8     ┆ 8                  │
│ 9     ┆ 9                  │
│ null  ┆ -1                 │
└───────┴────────────────────┘
```

Также вы можете воспользоваться одной из следующих стратегий заполнения пропусков, переданных аргументу `strategy`:

- `forward`: заполнение пропусков ближайшим предшествующим непропущенным значением;
- `backward`: заполнение пропусков ближайшим последующим непропущенным значением;
- `min`: заполнение минимальным значением в объекте Series;
- `max`: заполнение максимальным значением в объекте Series;
- `mean`: заполнение средним арифметическим значением в объекте Series. Обратите внимание, что в этом случае тип итогового значения будет приведен к типу данных объекта Series. Таким образом, если вы имеете дело с целочисленным столбцом, при усреднении будет отброшена дробная часть результата;
- `zero`: заполнение нулями;
- `one`: заполнение единицами.

Давайте в одном примере посмотрим на использование всех перечисленных стратегий:

```
missing_df.with_columns(
    forward=pl.col("value").fill_null(strategy="forward"),
    backward=pl.col("value").fill_null(strategy="backward"),
    min=pl.col("value").fill_null(strategy="min"),
    max=pl.col("value").fill_null(strategy="max"),
    mean=pl.col("value").fill_null(strategy="mean"),
```

```
    zero=pl.col("value").fill_null(strategy="zero"),
    one=pl.col("value").fill_null(strategy="one"),
)
```

Вывод:

```
shape: (10, 8)
```

| value | forward | backward | min | max | mean | zero | one |
|---|---|---|---|---|---|---|---|
| i64 | i64 | i64 | i64 | i64 | i64 | i64 | i64 |
| null | null | 2 | 2 | 9 | 5 | 0 | 1 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 | 4 | 4 | 4 | 4 |
| null | 4 | 7 | 2 | 9 | 5 | 0 | 1 |
| null | 4 | 7 | 2 | 9 | 5 | 0 | 1 |
| 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 |
| 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| null | 9 | null | 2 | 9 | 5 | 0 | 1 |

Третий способ заполнения пропущенных значений состоит в использовании выражения наподобие следующего: `pl.col("value").mean()`. Подробно о выражениях мы будем говорить в главе 7, здесь же просто посмотрим пример:

```
missing_df.with_columns(
    expression_mean=pl.col("value").fill_null(pl.col("value").mean())
)
```

Вывод:

```
shape: (10, 2)
```

| value | expression_mean |
|---|---|
| i64 | f64 |
| null | 5.5 |
| 2 | 2.0 |
| 3 | 3.0 |
| 4 | 4.0 |
| null | 5.5 |
| null | 5.5 |
| 7 | 7.0 |
| 8 | 8.0 |
| 9 | 9.0 |
| null | 5.5 |

В главе 8 мы рассмотрим и другие способы заполнения пропусков в данных. Пятый и последний способ, который мы продемонстрируем здесь, состоит в интерполяции с помощью метода *interpolate()*:

```
missing_df.interpolate()
```

Вывод:

```
shape: (10, 1)
┌────────┐
│ value  │
│ ---    │
│ f64    │
╞════════╡
│ null   │
│ 2.0    │
│ 3.0    │
│ 4.0    │
│ 5.0    │
│ 6.0    │
│ 7.0    │
│ 8.0    │
│ 9.0    │
│ null   │
└────────┘
```

### Числа с плавающей запятой: как они работают?

Тип данных *Float* используется для хранения чисел с плавающей запятой. При этом точность чисел характеризуется количеством битов, используемых для их представления. По умолчанию в Polars используется двойная точность, что соответствует 64 битам. Такая степень точности достигается при использовании типа данных `Float64`. Также библиотека поддерживает тип данных `Float32` для представления чисел с плавающей запятой одинарной точности с применением 32 бит. Использование этого типа позволяет сэкономить память, жертвуя при этом точностью.

Для описания принципов хранения чисел с плавающей запятой мы воспользуемся типом `Float32`. 32 имеющихся у нас в распоряжении бита хранят следующую информацию:

- первый бит представляет знак числа (0, если число положительное, 1 – если отрицательное);
- следующие восемь бит (со второго по девятый) представляют экспоненту, на которую умножается мантисса;
- оставшиеся 23 бита представляют мантиссу с неявной лидирующей единицей.

Формула для расчета значения с плавающей запятой выглядит так:

$$x = (-1)^{\text{знак}} * (1 + \text{мантисса}) * 2^{(\text{экспонента} - \text{смещение})}.$$

Смещение для типа данных `Float32` составляет 127. Таким образом, действительная степень двойки в десятичной форме получается путем вычитания этого смещения из двоичного представления экспоненты. Причина того, почему в числах с плавающей запятой используется смещение, заключается в том, что отрицательная сте-

пень, полученная в результате вычитания, позволяет эффективно представлять как очень большие, так и очень маленькие числа.

Давайте для примера рассмотрим следующее число с плавающей запятой в двоичном виде:

0 10000010 10100000000000000000000

где:

- 0 – означает, что число является положительным;
- 10000010 – экспонента в двоичном виде, соответствующая числу 130 в десятичном представлении;
- 10100000000000000000000 – мантисса в двоичной форме. Рассчитывается путем добавления неявной лидирующей единицы (для нормализованных чисел) к двоичным числам, что можно проинтерпретировать следующим образом: 1 (неявная лидирующая единица) + $1 * 2^{-1}$ (первое число, дающее в результате 0.5) + $0 * 2^{-2}$ (второе число, игнорируется из-за умножения на ноль) + $1 * 2^{-3}$ (третье число, дающее в результате 0.125). Оставшиеся числа – это нули, так что вклада в результат они не внесут. Таким образом, мантисса будет равна 1 + 0.5 + 0.125 = 1.625.

Подставив полученные значения в приведенную выше формулу, получим:

- $x = (-1)^0 * (1 + 0.5 + 0.125) * 2^{(130 - 127)}$;
- $x = 1 * 1.625 * 8$;
- $x = 13$.

# Преобразование типов данных

Часто бывает необходимо изменить тип данных столбца или объекта Series. К примеру, вы можете прочитать данные из файла CSV и решить, что одному из столбцов ошибочно был присвоен тип `String`, а на самом деле в нем хранятся числовые значения.

В таких случаях вы можете воспользоваться методом *`Expr.cast()`* или *`cast()`*.

С помощью метода `Expr.cast()` можно привести нужный вам столбец (чисто технически – выражение) к требуемому типу, передав его в качестве аргумента. Ниже показан пример, демонстрирующий важность выбора подходящих типов данных:

```
string_df = pl.DataFrame({"id": ["10000", "20000", "30000"]})
print(string_df)
print(f"Оценочный размер в байтах: {string_df.estimated_size('b')}")
```

Вывод:

```
shape: (3, 1)
┌───────┐
│ id    │
│ ---   │
│ str   │
╞═══════╡
┌───────┐
```

```
| 10000 |
| 20000 |
| 30000 |
└───────┘
Оценочный размер в байтах: 15
```

В то же время вы знаете, что в этом столбце хранятся исключительно числовые значения, которые можно хранить более эффективно. Изменить тип данных можно следующим образом:

```
int_df = string_df.select(pl.col("id").cast(pl.UInt16))
print(int_df)
print(f"Оценочный размер в байтах: {int_df.estimated_size('b')}")
```

Вывод:

```
shape: (3, 1)
┌───────┐
| id    |
| ---   |
| u16   |
╞═══════╡
| 10000 |
| 20000 |
| 30000 |
└───────┘
Оценочный размер в байтах: 6
```

Мы только что уменьшили объем используемой памяти более чем на 60 %. Применение оптимальных типов данных способно привести к существенной экономии ресурсов.

В табл. 4.1 мы также привели допустимые диапазоны для всех используемых в Polars типов данных. Чтобы максимально сэкономить на расходе памяти, необходимо приводить столбцы к минимально допустимому для хранения имеющихся данных типу.

В предыдущем примере мы воспользовались методом `Expr.cast()` для выражений. Также с этой целью можно применить и метод датафрейма `cast()`. В этом случае вы можете легко привести к нужному типу сразу несколько объектов Series, указав либо требуемый тип данных, либо словарь с парами из колонок и типов. В качестве ключей в словаре можно использовать либо имена столбцов, либо селекторы столбцов. Ниже показан пример применения метода `df.cast()` с приведением всех столбцов к одному типу:

```
data_types_df = pl.DataFrame(
    {
        "id": [10000, 20000, 30000],
        "value": [1.0, 2.0, 3.0],
        "value2": ["1", "2", "3"],
    }
)

data_types_df.cast(pl.UInt16)
```

Вывод:

```
shape: (3, 3)
┌─────────┬─────────┬─────────┐
│ id      ┆ value   ┆ value2  │
│ ---     ┆ ---     ┆ ---     │
│ u16     ┆ u16     ┆ u16     │
╞═════════╪═════════╪═════════╡
│ 10000   ┆ 1       ┆ 1       │
│ 20000   ┆ 2       ┆ 2       │
│ 30000   ┆ 3       ┆ 3       │
└─────────┴─────────┴─────────┘
```

А так можно выполнить приведение каждого объекта Series к своему типу данных:

```
data_types_df.cast({"id": pl.UInt16, "value": pl.Float32, "value2": pl.UInt8})
```

Вывод:

```
shape: (3, 3)
┌─────────┬─────────┬─────────┐
│ id      ┆ value   ┆ value2  │
│ ---     ┆ ---     ┆ ---     │
│ u16     ┆ f32     ┆ u8      │
╞═════════╪═════════╪═════════╡
│ 10000   ┆ 1.0     ┆ 1       │
│ 20000   ┆ 2.0     ┆ 2       │
│ 30000   ┆ 3.0     ┆ 3       │
└─────────┴─────────┴─────────┘
```

Вы также можете указать, какие типы данных на какие хотите поменять, как показано ниже:

```
data_types_df.cast({pl.Float64: pl.Float32, pl.String: pl.UInt8})
```

Вывод:

```
shape: (3, 3)
┌─────────┬─────────┬─────────┐
│ id      ┆ value   ┆ value2  │
│ ---     ┆ ---     ┆ ---     │
│ i64     ┆ f32     ┆ u8      │
╞═════════╪═════════╪═════════╡
│ 10000   ┆ 1.0     ┆ 1       │
│ 20000   ┆ 2.0     ┆ 2       │
│ 30000   ┆ 3.0     ┆ 3       │
└─────────┴─────────┴─────────┘
```

Наконец, вы можете воспользоваться селектором столбцов:

```
data_types_df.cast({cs.numeric(): pl.UInt16})
```

Вывод:

```
shape: (3, 3)
┌─────────┬─────────┬─────────┐
│ id      ┆ value   ┆ value2  │
│ ---     ┆ ---     ┆ ---     │
│ u16     ┆ u16     ┆ str     │
╞═════════╪═════════╪═════════╡
│ 10000   ┆ 1       ┆ 1       │
│ 20000   ┆ 2       ┆ 2       │
│ 30000   ┆ 3       ┆ 3       │
└─────────┴─────────┴─────────┘
```

В главе 10 мы подробнее поговорим о селекторах столбцов.

Но базовые методы приведения типов не всегда срабатывают так, как нам хотелось бы. Зачастую вкупе с ними приходится использовать особые методы для преобразования одних типов в другие. Ярким примером является извлечение дат из текстового содержимого. В главе 12 мы детально изучим тему преобразования типов.

# Заключение

В этой главе вы узнали:

- какие структуры данных используются в Polars. Основные из них – это объект Series, а также традиционный и ленивый датафреймы;
- какие типы данных применяются в Polars;
- о том, что некоторые типы данных используют свой набор операций. Примерами таких типов являются текстовый, тип для хранения даты и времени, а также вложенные типы. Подробнее об этом мы поговорим в главе 12;
- о способах обработки пропущенных значений в Polars;
- о приведении типов данных с помощью методов `Expr.cast()` и `df.cast()`.

Этой информации вам будет достаточно для создания и наполнения собственных датафреймов. В следующей главе мы с головой окунемся в мир API, предлагаемых библиотекой Polars.

# Глава 5

# Жадный и ленивый API

Теперь, когда вы знаете о структурах и типах данных, использующихся в Polars, посмотрим, какие *интерфейсы прикладного программирования*, или *API* (application programming interface), применяются в этой библиотеке для взаимодействия с данными. А интерфейсов всего два: жадный и ленивый. Каждый из этих интерфейсов используется для своих специфических задач и обладает своим набором сильных и слабых сторон. Хорошее понимание работы этих API является критически важным компонентом освоения библиотеки Polars.

В этой главе вы узнаете о том, что:

- в жадном API применяется немедленная модель выполнения, идеально подходящая для исследования данных и итеративных задач;
- в ленивом API операции преобразования данных откладываются до момента, когда их выполнение будет необходимо, что позволяет оптимизировать запросы и повышать быстродействие, особенно при работе с большими наборами данных;
- каждый API предназначен для своих задач, и важно правильно выбирать интерфейс в зависимости от сценария.

Все инструкции по загрузке нужных файлов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

## Жадный API: датафреймы

*Жадный API* (eager API) в Polars работает традиционным способом, по принципу немедленного выполнения. Все операции по работе с набором данных в этом случае выполняются последовательно. Такой подход может оказаться эффективным при исследовании данных и при выполнении итеративного анализа, поскольку позволяет взаимодействовать с данными на каждом шаге обработки. Вы можете применять функции к промежуточным результатам вычислений, что обеспечивает удобную обратную связь. Это бывает крайне

необходимо для оценки эффективности и необходимости выполнения следующих запросов. Жадный API в плане использования очень напоминает pandas, так что для плавного перехода разработчики часто выбирают именно его.

Ниже приведен пример использования жадного API Polars. У нас есть набор данных с поездками на такси, и нам необходимо проанализировать данные и выявить трех лидеров по отношению выручки к преодоленной дистанции. Мы разобьем процесс анализа на части, что позволит продемонстрировать пользу от применения жадного API. Обратите внимание, что в начале ячейки мы вставили магическую команду `%%time`, позволяющую измерить время выполнения операции:

```
%%time
trips = pl.read_parquet("data/taxi/yellow_tripdata_*.parquet") ❶
sum_per_vendor = trips.group_by("VendorID").sum() ❷

income_per_distance_per_vendor = sum_per_vendor.select(
    "VendorID",
    income_per_distance=pl.col("total_amount") / pl.col("trip_distance"),
)

top_three = income_per_distance_per_vendor.sort( ❸
    by="income_per_distance", descending=True
).head(3)

top_three
```

❶ Здесь мы читаем все файлы Parquet, удовлетворяющие заданной маске. *Маской* (glob pattern) называется строковое определение шаблона, используемого для выбора файлов. Подробнее о чтении и записи файлов мы будем говорить в главе 6. На данный момент будет достаточно знания о том, что наш набор данных состоит из нескольких файлов, которые мы загружаем в один датафрейм. Функция *`pl.read_parquet()`* возвращает датафрейм, полученный при помощи жадного API.

❷ Группируем данные по столбцу `VendorID`, что позволяет получить сумму по каждой группе.

❸ Вычисляем отношение выручки к преодоленной дистанции для каждой компании и оставляем трех лидеров.

Вывод:

```
CPU times: user 9.45 s, sys: 8.7 s, total: 18.1 s
Wall time: 8.52 s
shape: (3, 2)
┌──────────┬─────────────────────┐
│ VendorID ┆ income_per_distance │
│ ---      ┆ ---                 │
│ i64      ┆ f64                 │
╞══════════╪═════════════════════╡
│ 1        ┆ 6.434789            │
│ 6        ┆ 5.296493            │
│ 5        ┆ 4.731557            │
└──────────┴─────────────────────┘
```

Выполнение этого фрагмента кода позволило нам ответить на поставленный вопрос. При выполнении подобного анализа бывает полезно разбивать общую задачу на более мелкие части. Так вы сможете анализировать промежуточные данные на каждом шаге, что поможет определиться с необходимостью выполнять последующие операции.

# Ленивый API: ленивые датафреймы

*Ленивый API* (lazy API) откладывает все операции по отбору, фильтрации и манипуляции данными до момента, когда их выполнение будет необходимо. Это позволяет движку запросов собрать всю необходимую информацию о том, какие операции и в какой последовательности в действительности нужно будет выполнить, и предпринять действия по оптимизации, направленные на ускорение выполнения запроса. Применение ленивого API бывает оправдано при работе с большими и сложными наборами данных и выполнении критических с точки зрения быстродействия операций.

Оптимизации, применяющиеся в процессе использования ленивого API, направлены в первую очередь на снижение объема данных, необходимых для обработки, а также на минимизацию количества операций, в действительности требующихся для достижения желаемого результата. Снижение объема обрабатываемых данных в том числе достигается за счет того, что ненужная информация, выявляемая на этапе сканирования запроса, даже не считывается из источника. Для этого предпринимаются следующие действия:

- считываются только нужные столбцы;
- заранее отфильтровываются строки, которые нам не понадобятся;
- в запросах участвуют только те части столбцов, которые действительно нуждаются в обработке.

Все эти действия выполняются под капотом при помощи планировщика и оптимизатора запросов. Подробнее об оптимизациях, используемых в ленивом API, мы будем говорить в главе 18. Главное, что вы должны усвоить о ленивом API, – это то, что он сначала думает, потом делает.

Кроме того, ленивый API способен отлавливать ошибки, связанные с типами данных, перед выполнением запроса (также эти ошибки называются *ошибками схемы* (SchemaError)). В плане выполнения запроса содержится информация о том, что должно происходить на каждом шаге и как должен выглядеть результат.

Рассмотрим следующий пример. Мы создадим ленивый датафрейм, содержащий имена и возраст трех людей. Если попытаться взять объект Series с именем `age`, содержащий данные типа `Int64`, и применить к нему строковые операции, мы моментально получим ошибку, еще до выполнения запроса:

```
names_lf = pl.LazyFrame(
    {"name": ["Alice", "Bob", "Charlie"], "age": [25, 30, 35]}
)
```

```
erroneous_query = names_lf.with_columns(
    sliced_age=pl.col("age").str.slice(1, 3)
)

result_df = erroneous_query.collect()
```

Вывод:

```
SchemaError: invalid series dtype: expected `String`, got `i64` for series with name `age`
```

Это позволяет экономить время и повысить эффективность процесса создания программного кода. Бывает, что при работе с большими наборами данных и объемными запросами вы получаете ошибку ближе к концу их выполнения. Применение ленивого API позволит не тратить время впустую.

# Разница в быстродействии

Лучший способ сравнить быстродействие разных API – применить их к одним и тем же данным и операциям.

```
%%time
trips = pl.scan_parquet("data/taxi/yellow_tripdata_*.parquet")
sum_per_vendor = trips.group_by("VendorID").sum()

income_per_distance_per_vendor = sum_per_vendor.select(
    "VendorID",
    income_per_distance=pl.col("total_amount") / pl.col("trip_distance"),
)

top_three = income_per_distance_per_vendor.sort(
    by="income_per_distance", descending=True
).head(3)

top_three.collect()
```

Вывод:

```
CPU times: user 2.01 s, sys: 301 ms, total: 2.31 s
Wall time: 592 ms
shape: (3, 2)
┌──────────┬─────────────────────┐
│ VendorID ┆ income_per_distance │
│ ---      ┆ ---                 │
│ i64      ┆ f64                 │
╞══════════╪═════════════════════╡
│ 1        ┆ 6.434789            │
│ 6        ┆ 5.296493            │
│ 5        ┆ 4.731557            │
└──────────┴─────────────────────┘
```

В результате мы получили тот же датафрейм, но ленивый API справился в десять раз быстрее жадного. Это ли не чудо?

В Polars ленивый датафрейм материализуется, или преобразуется в обычный датафрейм, только в момент вызова метода *collect()*. Хотя мы видим, какой прирост быстродействия обеспечил нам ленивый датафрейм, стоит учитывать, что при повторном вызове метода `collect()` он будет вычисляться заново. Это означает, что все операции будут запущены заново, чего нам бы хотелось избежать.

Давайте создадим маленький ленивый датафрейм с двумя столбцами и тремя строками, но поработаем с ним как с большим набором данных, требующим сложных вычислений:

```
lf = pl.LazyFrame({"col1": [1, 2, 3], "col2": [4, 5, 6]})

# ... Сложные вычисления ...

print(lf.collect())

print(lf.with_columns(pl.col("col1") + 1).collect()) ❶
```

❶ В этот момент ленивый датафрейм будет вычисляться заново.

Вывод:

```
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 1    ┆ 4    │
│ 2    ┆ 5    │
│ 3    ┆ 6    │
└──────┴──────┘
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 2    ┆ 4    │
│ 3    ┆ 5    │
│ 4    ┆ 6    │
└──────┴──────┘
```

В конце этой главы мы покажем вам трюк с кешированием, позволяющий избежать повторных вычислений.

# Разница в функционале

Основное отличие датафрейма от его ленивого аналога состоит в том, что во втором случае данные не будут доступны вплоть до момента фактического

вычисления. Это означает, что и функционал нам будет доступен не весь. В следующих разделах мы перечислим разные типы операций и посмотрим на различия в отношении них.

## Атрибуты

Из всех атрибутов, доступных для классических датафреймов, ленивые датафреймы не имеют доступа только к атрибутам *shape*, *height* и *flags*, что видно в табл. 5.1. Первые два связаны с размером датафрейма, который станет известен только при получении данных, а атрибут `flags` представляет собой словарь с индикаторами, такими как признак сортировки объекта Series, который используется движком для оптимизации.

***Таблица 5.1. Атрибуты датафреймов***

| Атрибут | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.columns` | ✓ | ✓ |
| `.dtypes` | ✓ | ✓ |
| `.flags` | ✓ | |
| `.height` | ✓ | |
| `.schema` | ✓ | ✓ |
| `.shape` | ✓ | |
| `.width` | ✓ | ✓ |

## Методы агрегации

В табл. 5.2 перечислены методы *вертикальной* (vertical aggregation) и *горизонтальной агрегации* (horizontal aggregation). Имена методов горизонтальной агрегации оканчиваются на `_horizontal`.

Все методы вертикальной агрегации, такие как *max()*, могут применяться как к традиционным, так и к ленивым датафреймам. Эти методы не требуют от движка запросов наличия предварительных знаний о данных и добавляются к плану запроса для применения к собранным данным.

Методы горизонтальной агрегации, такие как *sum_horizontal()*, применяются к строкам целиком и рассчитываются на основе столбцов. Эти методы предполагают наличие информации о данных, в связи с чем они доступны только для обычных датафреймов.

***Таблица 5.2. Методы агрегации***

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.count()` | ✓ | ✓ |
| `.max()` | ✓ | ✓ |
| `.max_horizontal()` | ✓ | |

***Таблица 5.2*** *(окончание)*

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.mean()` | ✓ | ✓ |
| `.mean_horizontal()` | ✓ | |
| `.median()` | ✓ | ✓ |
| `.min()` | ✓ | ✓ |
| `.min_horizontal()` | ✓ | |
| `.product()` | ✓ | |
| `.quantile()` | ✓ | ✓ |
| `.std()` | ✓ | ✓ |
| `.sum()` | ✓ | ✓ |
| `.sum_horizontal()` | ✓ | |
| `.var()` | ✓ | ✓ |

Метод *`product()`* также доступен только для классических датафреймов.

## Вычислительные методы

Обычные датафреймы содержат два *вычислительных метода* (computation method) *`fold()`* и *`hash_rows()`*, тогда как у ленивых датафреймов вычислительные методы отсутствуют. Оба перечисленных метода выполняются построчно и уменьшают данные. Метод `fold()` позволяет заданным способом свести два столбца в один, а метод `hash_rows()` кеширует информацию по строке в значение с типом данных `UInt64`.

## Описательные методы

К числу *описательных методов* (descriptive method), присущих ленивым датафреймам, но отсутствующих в арсенале обычных датафреймов, относятся методы *`explain()`* и *`show_graph()`* для прорисовки плана запроса, что видно в табл. 5.3. В то же время классические датафреймы обладают большим набором методов для описания своего содержимого, в числе которых методы *`describe()`* и *`estimated_size()`*.

***Таблица 5.3. Описательные методы***

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.approx_n_unique()` | ✓ | |
| `.describe()` | ✓ | [1] |
| `.estimated_size()` | ✓ | |
| `.explain()` | | ✓ |

[1] Чисто технически у ленивых датафреймов есть метод `describe()`, но он выполняет сбор результатов для анализа.

***Таблица 5.3*** *(окончание)*

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.glimpse()` | ✓ | |
| `.is_duplicated()` | ✓ | |
| `.is_empty()` | ✓ | |
| `.is_unique()` | ✓ | |
| `.n_chunks()` | ✓ | |
| `.n_unique()` | ✓ | |
| `.null_count()` | ✓ | ✓ |
| `.show_graph()` | | ✓ |

# Методы группировки

Все методы, применяющиеся к группам в контексте *объекта GroupBy*, у классических и ленивых датафреймов одинаковые, за исключением того, что в первом случае вы можете проходить итерациями по группам. Это отражено в табл. 5.4.

***Таблица 5.4. Методы группировки***

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.__iter__()` | ✓ | |
| `.agg()` | ✓ | ✓ |
| `.all()` | ✓ | ✓ |
| `.count()` | ✓ | ✓ |
| `.first()` | ✓ | ✓ |
| `.head()` | ✓ | ✓ |
| `.last()` | ✓ | ✓ |
| `.len()` | ✓ | ✓ |
| `.map_groups()` | ✓ | ✓ |
| `.max()` | ✓ | ✓ |
| `.mean()` | ✓ | ✓ |
| `.median()` | ✓ | ✓ |
| `.min()` | ✓ | ✓ |
| `.n_unique()` | ✓ | ✓ |
| `.quantile()` | ✓ | ✓ |
| `.sum()` | ✓ | ✓ |
| `.tail()` | ✓ | ✓ |

# Методы экспорта

Датафреймы содержат несколько методов для сохранения данных в различных форматах (см. табл. 5.5). Поскольку ленивые датафреймы не содержат данных, они лишены каких бы то ни было методов для экспорта.

*Таблица 5.5. Методы экспорта*

| Метод | Описание |
|---|---|
| `.to_arrow()` | Собирает лежащие в основе датафрейма массивы Arrow в таблицу Arrow |
| `.to_dict()` | Преобразует датафрейм в словарь с именами объектов Series в качестве ключей и их содержимым в качестве значений |
| `.to_dicts()` | Преобразует каждую строку датафрейма в словари Python |
| `.to_init_repr()` | Преобразует датафрейм в строковое представление |
| `.to_jax()` | Преобразует датафрейм в массив JAX или словарь из массивов JAX |
| `.to_numpy()` | Преобразует датафрейм в массив Numpy |
| `.to_pandas()` | Преобразует датафрейм в датафрейм pandas |
| `.to_struct()` | Преобразует датафрейм в объект Series из типов Struct |
| `.to_torch()` | Преобразует датафрейм в тензор PyTorch, набор или словарь тензоров |

Еще датафреймы и ленивые датафреймы обладают методом *`serialize()`*, который также может быть причислен к методам экспорта.

# Методы манипуляции и отбора данных

*Методы манипуляции и отбора данных* представляют собой важнейшую группу методов, включающую все наиболее часто используемые операции. В табл. 5.6 показано, какие методы в каких датафреймах присутствуют.

*Таблица 5.6. Методы манипуляции и отбора данных*

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.bottom_k()` | ✓ | ✓ |
| `.cast()` | ✓ | ✓ |
| `.clear()` | ✓ | ✓ |
| `.clone()` | ✓ | ✓ |
| `.drop()` | ✓ | ✓ |
| `.drop_in_place()` | ✓ | |
| `.drop_nulls()` | ✓ | ✓ |
| `.explode()` | ✓ | ✓ |
| `.extend()` | ✓ | |
| `.fill_nan()` | ✓ | ✓ |
| `.fill_null()` | ✓ | ✓ |
| `.filter()` | ✓ | ✓ |
| `.first()` | | ✓ |
| `.gather_every()` | ✓ | ✓ |
| `.get_column()` | ✓ | |
| `.get_column_index()` | ✓ | |
| `.get_columns()` | ✓ | |
| `.group_by()` | ✓ | ✓ |

***Таблица 5.6*** *(продолжение)*

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.group_by_dynamic()` | ✓ | ✓ |
| `.head()` | ✓ | ✓ |
| `.hstack()` | ✓ | |
| `.insert_column()` | ✓ | |
| `.inspect()` | | ✓ |
| `.interpolate()` | ✓ | ✓ |
| `.item()` | ✓ | |
| `.iter_columns()` | ✓ | |
| `.iter_rows()` | ✓ | |
| `.iter_slices()` | ✓ | |
| `.join()` | ✓ | ✓ |
| `.join_asof()` | ✓ | ✓ |
| `.join_where()` | ✓ | ✓ |
| `.last()` | | ✓ |
| `.limit()` | ✓ | ✓ |
| `.melt()` | ✓ | ✓ |
| `.merge_sorted()` | ✓ | ✓ |
| `.partition_by()` | ✓ | |
| `.pipe()` | ✓ | |
| `.pivot()` | ✓ | |
| `.rechunk()` | ✓ | |
| `.rename()` | ✓ | ✓ |
| `.replace_column()` | ✓ | |
| `.reverse()` | ✓ | ✓ |
| `.rolling()` | ✓ | ✓ |
| `.row()` | ✓ | |
| `.rows()` | ✓ | |
| `.rows_by_key()` | ✓ | |
| `.sample()` | ✓ | |
| `.select()` | ✓ | ✓ |
| `.select_seq()` | ✓ | ✓ |
| `.set_sorted()` | ✓ | ✓ |
| `.shift()` | ✓ | ✓ |
| `.shrink_to_fit()` | ✓ | |
| `.slice()` | ✓ | ✓ |
| `.sort()` | ✓ | ✓ |
| `.sql()` | ✓ | ✓ |
| `.tail()` | ✓ | ✓ |
| `.to_dummies()` | ✓ | |
| `.to_series()` | ✓ | |
| `.top_k()` | ✓ | ✓ |
| `.transpose()` | ✓ | |
| `.unique()` | ✓ | ✓ |
| `.unnest()` | ✓ | ✓ |

***Таблица 5.6*** *(окончание)*

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.unpivot()` | ✓ | ✓ |
| `.unstack()` | ✓ | |
| `.update()` | ✓ | ✓ |
| `.upsample()` | ✓ | |
| `.vstack()` | ✓ | |
| `.with_columns()` | ✓ | ✓ |
| `.with_columns_seq()` | ✓ | ✓ |
| `.with_context()` | | ✓ |
| `.with_row_count()` | ✓ | ✓ |
| `.with_row_index()` | ✓ | ✓ |

# Прочие методы

В табл. 5.7 перечислены методы, не вошедшие ни в одну из этих категорий.

***Таблица 5.7. Прочие методы***

| Метод | Датафрейм | Ленивый датафрейм |
|---|---|---|
| `.cache()` | | ✓ |
| `.collect()` | | ✓ |
| `.collect_async()` | | ✓ |
| `.collect_schema()` | ✓ | ✓ |
| `.corr()` | ✓ | |
| `.equals()` | ✓ | |
| `.lazy()` | ✓ | ✓ |
| `.map_batches()` | | ✓ |
| `.map_rows()` | ✓ | |
| `.pipe()` | | ✓ |
| `.profile()` | | ✓ |

### Вычисления с использованием внешней памяти

Ленивый API также предлагает особый режим вычислений с *использованием внешней памяти* (out of core), заключающийся в обработке данных, не помещающихся целиком в оперативную память, порциями. Прелесть этого режима состоит в возможности задействования места на жестком диске, которого обычно в разы больше, чем способна вместить оперативная память. Активировать этот режим можно, передав параметр `streaming=True` методу `collect()` для сбора окончательного результата в оперативную память, или вы можете воспользоваться записью результатов на диск с помощью методов *`sink_csv()`*, *`sink_ipc()`*, *`sink_parquet()`* или *`sink_ndjson()`*. При использовании вызова `lf.collect(streaming=True)` итоговый результат должен помещаться в оперативную память.

В потоковом режиме API читает данные блоками строк. Размер блока определяется Polars исходя из возможностей вашего аппаратного обеспечения и набора данных, с которым вы работаете. Этот API используется так же точно, как ленивый API, отличается лишь модель вычисления под капотом.

На момент написания книги этот режим является экспериментальным, и мы не будем касаться его в данном издании.

# Полезные советы

В этом разделе мы рассмотрим несколько важных нюансов при работе с датафреймами. Многие из них могут оказаться весьма полезными в вашей ежедневной работе с Polars. Это та информация, которую вы вряд ли почерпнете из документации, но которая способна существенно облегчить вам жизнь.

## Переход от ленивых датафреймов к обычным и обратно

Вы можете переключаться с одного API на другой при помощи простых методов, показанных на рис. 5.1.

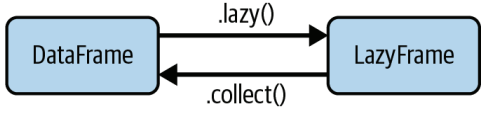



**Рис. 5.1** ❖ Методы для переключения между API

Для перехода от жадного API к ленивому вам достаточно вызвать метод *`lazy()`* у датафрейма или подставить его в конец цепочки методов, возвращающих датафрейм. Это приведет к тому, что никакие вычисления производиться не будут, а планировщик будет готов использовать данные, собранные в памяти, для построения плана выполнения следующих запросов.

Чтобы перейти обратно от ленивого API к жадному, вам необходимо применить метод *`collect()`* к ленивому датафрейму или к функции, возвращающей ленивый датафрейм. Это повлечет вычисление ранее построенного для этого датафрейма плана. После этого результаты будут сохранены в памяти. Если вы используете потоковый режим и вместо метода `collect()` вызываете метод *`sink_parquet()`*, результаты будут записаны на диск, что позволит избежать ошибок, связанных с нехваткой памяти.

# Объединение обычных и ленивых датафреймов

При выполнении операций объединения в Polars используемые структуры данных должны иметь один тип. Таким образом, вы не можете напрямую объединить ленивый датафрейм с обычным. В то же время это может быть полезно, если у вас, например, есть небольшой датафрейм с метаданными, который вы хотите объединить с большим датафреймом, использующим ленивую парадигму.

Приведенный ниже фрагмент кода выдаст ошибку:

```
big_sales_data = pl.LazyFrame(
    {"sale_id": [101, 102, 103], "amount": [250, 150, 300]}
)

sales_metadata = pl.DataFrame(
    {"sale_id": [101, 102, 103], "category": ["A", "B", "A"]}
)

big_sales_data.join(sales_metadata, on="sale_id").collect()
```

Вывод:

```
TypeError: expected `other` join table to be a LazyFrame, not a 'DataFrame'
```

К счастью, есть простой способ обойти эту проблему. Вы можете предварительно либо сделать свой датафрейм с метаданными ленивым с помощью метода `lazy()`, либо материализовать ленивый датафрейм, воспользовавшись методом `collect()`. С целью повышения быстродействия мы рекомендуем отдавать предпочтение ленивой парадигме вычислений.

Ниже показано, как можно объединить два датафрейма разных типов:

```
big_sales_data = pl.LazyFrame(
    {"sale_id": [101, 102, 103], "amount": [250, 150, 300]}
)

sales_metadata = pl.DataFrame(
    {"sale_id": [101, 102, 103], "category": ["A", "B", "A"]}
)

big_sales_data.join(sales_metadata.lazy(), on="sale_id").collect()
```

Вывод:

```
shape: (3, 3)
┌─────────┬────────┬──────────┐
│ sale_id ┆ amount ┆ category │
│ ---     ┆ ---    ┆ ---      │
│ i64     ┆ i64    ┆ str      │
╞═════════╪════════╪══════════╡
├─────────┼────────┼──────────┤
│ 101     ┆ 250    ┆ A        │
```

```
| 102     | 150    | B       |
| 103     | 300    | A       |
└─────────┴────────┴─────────┘
```

Если в первом случае вы получили ошибку, то сейчас у вас есть полноценный ленивый датафрейм.

## Кеширование промежуточных результатов

Ленивые датафреймы – это круто, но до настоящей магии им, увы, далеко. В частности, они обладают одним неудобным свойством, заключающимся в том, что если вы вычислили ленивый датафрейм и не сохранили его в переменной в памяти или на диске, при следующем обращении к нему он будет вычисляться заново. Во избежание лишних вычислений вы можете воспользоваться операцией *кеширования ленивых датафреймов*. Кеширование заключается в вызове цепочки методов `collect().lazy()` после выполнения ресурсоемких операций и сохранении результата в переменной. Это приведет к вычислению ленивого датафрейма, сохранению его в памяти и возвращению нового ленивого датафрейма, указывающего на материализованные данные в оперативной памяти.

Вот как вы можете оптимизировать предыдущий пример. В данном случае нас не беспокоит время выполнения, мы лишь пытаемся донести до вас общую идею.

```
lf = pl.LazyFrame({"col1": [1, 2, 3], "col2": [4, 5, 6]})

# ... Сложные вычислительные операции ...

lf = lf.collect().lazy() ❶
print(lf.collect())

print(lf.with_columns(pl.col("col1") + 1).collect()) ❷
```

❶ Кешируем ленивый датафрейм.
❷ Используем кешированный датафрейм.

Вывод:

```
shape: (3, 2)
┌──────┬──────┐
| col1 | col2 |
| ---  | ---  |
| i64  | i64  |
╞══════╪══════╡
| 1    | 4    |
| 2    | 5    |
| 3    | 6    |
└──────┴──────┘
```

```
shape: (3, 2)
┌──────┬──────┐
│ col1 ┆ col2 │
│ ---  ┆ ---  │
│ i64  ┆ i64  │
╞══════╪══════╡
│ 2    ┆ 4    │
│ 3    ┆ 5    │
│ 4    ┆ 6    │
└──────┴──────┘
```

Если бы мы не кешировали ленивый датафрейм, второй вызов метода `collect()` привел бы к его повторному вычислению с нуля. Этот прием может быть весьма полезен при работе с объемными вычислительными операциями, поскольку позволяет воспользоваться всеми преимуществами ленивых вычислений, сводя на нет их недостатки.

# Заключение

В этой главе мы познакомились с жадным и ленивым API в Polars. Помимо прочего, вы узнали, что:

- жадный API предназначен для немедленного выполнения запросов и бывает наиболее полезен при выполнении итеративных вычислений и исследовании данных;
- ленивый API использует концепцию отложенных вычислений с целью предварительной оптимизации построенных планов выполнения запросов. Его стоит применять в случаях, когда критически важным является время выполнения операций, а промежуточные результаты в процессе вычислений не используются;
- ленивые датафреймы не содержат данных, тогда как в обычных датафреймах данные содержатся, что обусловливает некоторую разницу в их функционале;
- потоковый режим работы с ленивым API позволяет выполнять вычисления с использованием внешней памяти при взаимодействии с данными, не помещающимися целиком в оперативную память;
- ленивые и обычные датафреймы можно объединять с предварительным преобразованием из одного API в другой, а также использовать операцию кеширования во избежание повторных вычислений ленивых датафреймов.

Полученных знаний вам должно быть достаточно для определения того, какой API в какой ситуации лучше использовать. Теперь пришло время поговорить о загрузке данных из файлов в структуры, которые мы обсуждали в этой главе. Так что следующую главу мы посвятим чтению и записи в разные форматы файлов.

# Глава 6

# Чтение и запись данных

Теперь, когда вы познакомились с некоторыми ключевыми концепциями Polars, такими как типы данных и разные API, вы вполне готовы к двустороннему взаимодействию с внешними источниками данных, включая файлы различных форматов и базы данных. Прочитав эту главу, вы сможете начать работать с вашими собственными данными. И чем раньше это произойдет, тем лучше, поскольку только так вы сможете осознать всю мощь и потенциал библиотеки Polars.

Внешние данные могут содержаться в самых разных форматах и поступать отовсюду. Именно поэтому в Polars предусмотрено порядка 30 функций и методов для чтения данных из разных форматов, и они принимают большое количество аргументов. Конечно, было бы очень утомительно и скучно подробно описывать все подобные функции и методы в этой главе. В конце концов, для этого существует официальная документация. Мы лишь бегло расскажем обо всех форматах, поддерживаемых Polars, а также познакомим вас с методами для работы с ними. Упор мы будем делать на наиболее полезных аргументах, с которыми вы будете работать на ежедневной основе.

В этой главе вы узнаете, как:

- читать и записывать данные в разных форматах, включая CSV, Excel и Parquet;
- одновременно работать с несколькими файлами, используя шаблоны;
- корректно обрабатывать при чтении пропущенные значения;
- эффективно работать с разными кодировками;
- читать данные с использованием жадной и ленивой парадигмы.

Для работы с файлами в разных форматах мы будем использовать следующие дополнительные пакеты:

- xlsx2csv для чтения файлов в формате Excel;
- chardet для определения кодировки файлов;
- ConnectorX для подключения к базам данных;
- pyarrow для чтения наборов данных PyArrow.

Все инструкции по загрузке нужных файлов и установке пакетов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

# Обзор форматов файлов

Библиотека Polars поддерживает большое количество форматов файлов. В табл. 6.1 приведены поддерживаемые форматы и соответствующие функции и методы для чтения и записи. Для краткости мы не стали указывать префиксы `pl.` и `df.` в таблице. Имена, начинающиеся с `read_` и `scan_`, принадлежат функциям Polars верхнего уровня, а имена, начинающиеся с `write_` и `sink_`, указывают на методы обычных и ленивых датафреймов.

***Таблица 6.1. Поддерживаемые Polars форматы ввода/вывода и соответствующие им функции чтения и методы записи***

| Формат | Чтение (pl.) | Чтение в ленивом режиме (pl.) | Запись (df.) | Запись в потоковом режиме (df.) |
|---|---|---|---|---|
| Avro | *pl.read_avro()* | | *write_avro()* | |
| Буфер обмена | *pl.read_clipboard()* | | *write_clipboard()* | |
| CSV | *pl.read_csv()*<br>*pl.read_csv_batched()* | *pl.scan_csv()* | *write_csv()* | *sink_csv()* |
| Базы данных | *pl.read_database()*<br>*pl.read_database_uri()* | | *write_database()* | |
| Delta Lake | *pl.read_delta()* | *pl.scan_delta()* | *write_delta()* | |
| Excel/ODS | *pl.read_excel()*<br>*pl.read_ods()* | | *write_excel()* | |
| Feather/IPC | *pl.read_ipc()*<br>*pl.read_ipc_schema()*<br>*pl.read_ipc_stream()* | *pl.scan_ipc()* | *write_ipc()*<br>*write_ipc_stream()* | *sink_ipc()* |
| Iceberg | | *pl.scan_iceberg()* | | |
| JSON | *pl.read_json()*<br>*pl.read_ndjson()* | *pl.scan_ndjson()* | *write_json()*<br>*write_ndjson()* | *sink_ndjson()* |
| Parquet | *pl.read_parquet()*<br>*pl.read_parquet_schema()* | *pl.scan_parquet()* | *write_parquet()* | *sink_parquet()* |
| Наборы данных PyArrow[1] | | *pl.scan_pyarrow_dataset()* | | |

# Чтение файлов CSV

Начнем с, пожалуй, самого распространенного в области программирования и анализа данных формата файлов с разделителями в виде запятых – *CSV*. Несмотря на свое широкое распространение, этот формат не лишен недо-

[1] Фактически это не формат файла. Polars может использовать движок PyArrow для разбора данных, соответствующих формату, который может прочитать pyarrow.

статков. Дело в том, что, получая в распоряжение файл в формате CSV, мы никогда не знаем, что в нем на самом деле окажется:

- какой символ используется в качестве разделителя: запятая, табуляция, точка с запятой или что-то еще;
- какая кодировка используется в файле: UTF-8, ASCII или какая-то другая;
- присутствуют ли в файле заголовки, и если да, то на скольких строках они располагаются;
- как в данных представлены пропущенные значения;
- используются ли в данных кавычки для обрамления строковых значений.

В библиотеке Polars присутствуют инструменты для помощи в ответе на эти вопросы, но бывают самые разные ситуации, в связи с чем чтение из подобных файлов необходимо выполнять с осторожностью.

Допустим, у нас есть файл *data/penguins.csv*. Перед загрузкой данных из него в Polars давайте взглянем на его содержимое с помощью утилиты командной строки *cat*[1]:

```
! cat data/penguins.csv
```

Вывод:

```
"rowid","species","island","bill_length_mm","bill_depth_mm","flipper_length_mm"...
"1","Adelie","Torgersen",39.1,18.7,181,3750,"male",2007
"2","Adelie","Torgersen",39.5,17.4,186,3800,"female",2007
"3","Adelie","Torgersen",40.3,18,195,3250,"female",2007
"4","Adelie","Torgersen",NA,NA,NA,NA,NA,2007
"5","Adelie","Torgersen",36.7,19.3,193,3450,"female",2007
"6","Adelie","Torgersen",39.3,20.6,190,3650,"male",2007
"7","Adelie","Torgersen",38.9,17.8,181,3625,"female",2007
"8","Adelie","Torgersen",39.2,19.6,195,4675,"male",2007
"9","Adelie","Torgersen",34.1,18.1,193,3475,NA,2007
... with 335 more lines
```

На первый взгляд с файлом все в порядке. Первая строка содержит заголовки столбцов, а в качестве разделителей используются запятые, что соответствует настройкам Polars по умолчанию. Кроме того, кодировка файла совместима с UTF-8 (подробнее об этом мы будем говорить далее). Это дает нам уверенность в том, что чтение данных из этого файла произойдет успешно:

```
penguins = pl.read_csv("data/penguins.csv")
penguins
```

Вывод:

```
shape: (344, 9)
┌───────┬─────────┬──────────┬───┬─────────────┬────────┬──────┐
│ rowid ┆ species ┆ island   ┆ … ┆ body_mass_g ┆ sex    ┆ year │
```

[1] Если вы используете Windows и команда `cat` вам недоступна, вы можете вместо нее воспользоваться командой `type`.

```
| ---   | ---       | ---       |     | ---         | ---    | ---  |
| i64   | str       | str       |     | str         | str    | i64  |
╞═══════╪═══════════╪═══════════╪═════╪═════════════╪════════╪══════╡
| 1     | Adelie    | Torgersen | …   | 3750        | male   | 2007 |
| 2     | Adelie    | Torgersen | …   | 3800        | female | 2007 |
| 3     | Adelie    | Torgersen | …   | 3250        | female | 2007 |
| 4     | Adelie    | Torgersen | …   | NA          | NA     | 2007 |
| 5     | Adelie    | Torgersen | …   | 3450        | female | 2007 |
| …     | …         | …         | …   | …           | …      | …    |
| 340   | Chinstrap | Dream     | …   | 4000        | male   | 2009 |
| 341   | Chinstrap | Dream     | …   | 3400        | female | 2009 |
| 342   | Chinstrap | Dream     | …   | 3775        | male   | 2009 |
| 343   | Chinstrap | Dream     | …   | 4100        | male   | 2009 |
| 344   | Chinstrap | Dream     | …   | 3775        | female | 2009 |
└───────┴───────────┴───────────┴─────┴─────────────┴────────┴──────┘
```

Вроде все нормально, за исключением того, что пропущенные значения были вставлены в ячейки в виде строк `NA`, что можно наблюдать, например, в четвертой строке в столбцах `body_mass_g` и `sex`. Скоро мы это поправим.

Если с вашим файлом CSV проблем еще больше, вам могут пригодиться аргументы, приведенные в табл. 6.2.

***Таблица 6.2. Часто используемые аргументы функции*** *`pl.read_csv()`*

| Аргумент | Описание |
|---|---|
| `Source` | Путь к файлу или объекту |
| `has_header` | Индикатор того, содержит ли первая строка файла заголовки столбцов |
| `Columns` | Столбцы для выбора. Принимает список индексов столбцов (начиная с нуля) или имен столбцов |
| `Separator` | Однобайтовый символ, использующийся в качестве разделителя в файле |
| `skip_rows` | Пропуск определенного количества строк |
| `null_values` | Значения, которые необходимо интерпретировать как пропущенные |
| `Encoding` | Кодировка. Значение по умолчанию – `utf8`. Значение `utf8-lossy` говорит о замене неизвестных символов на специальный символ `�`. При использовании отличных от utf8 и utf8-lossy кодировок входящие данные сначала декодируются в памяти средствами Python |

# Корректная обработка пропущенных значений

В наборах данных довольно часто присутствуют пропущенные значения. К сожалению, в плоских форматах, коим является CSV, отсутствует единый способ представления пропусков в данных. Какие только варианты нам не встречались в «дикой природе» – NULL, Nil, None, NA, N/A, NaN, 999999 и пустые строки.

По умолчанию Polars интерпретирует пустые строки в данных как пропущенные значения. Любые другие значения, которые необходимо воспри-

нимать как пропуски, необходимо в явном виде передавать как строку или список строк аргументу `null_values`. Давайте исправим нашу оплошность при чтении данных из файла *data/penguins.csv*:

```
penguins = pl.read_csv("data/penguins.csv", null_values="NA")
penguins
```

Вывод:

```
shape: (344, 9)
```

| rowid | species | island | … | body_mass_g | sex | year |
|---|---|---|---|---|---|---|
| i64 | str | str | | i64 | str | i64 |
| 1 | Adelie | Torgersen | … | 3750 | male | 2007 |
| 2 | Adelie | Torgersen | … | 3800 | female | 2007 |
| 3 | Adelie | Torgersen | … | 3250 | female | 2007 |
| 4 | Adelie | Torgersen | … | null | null | 2007 |
| 5 | Adelie | Torgersen | … | 3450 | female | 2007 |
| … | … | … | … | … | … | … |
| 340 | Chinstrap | Dream | … | 4000 | male | 2009 |
| 341 | Chinstrap | Dream | … | 3400 | female | 2009 |
| 342 | Chinstrap | Dream | … | 3775 | male | 2009 |
| 343 | Chinstrap | Dream | … | 4100 | male | 2009 |
| 344 | Chinstrap | Dream | … | 3775 | female | 2009 |

**Датафреймы в книге и в Jupyter Notebook**

При представлении датафреймов в стандарте кодирования ASCII, как в этой книге, все строковые значения выводятся без кавычек. Это может помешать визуально убедиться в том, что все пропущенные значения были проинтерпретированы правильно. При использовании Jupyter Notebook датафреймы выводятся в виде представления HTML, как мы видели на рис. 3.2. В таком виде пропуски отображаются как `null` без кавычек, тогда как строковые значения выводятся с кавычками.

Если вы не уверены, были ли в процессе чтения данных правильно проинтерпретированы пропущенные значения, вы можете проверить наличие пропусков в датафрейме программно, с помощью метода *`null_count()`*, как показано ниже:

```
penguins.null_count().transpose( ❶
    include_header=True, column_names=["null_count"]
)
```

❶ Мы транспонировали результат, чтобы можно было более наглядно отследить все пропуски в данных.

Вывод:

```
shape: (9, 2)
```

| column | null_count |
|---|---|

```
| ---                | ---       |
| str                | u32       |
╞════════════════════╪═══════════╡
| rowid              | 0         |
| species            | 0         |
| island             | 0         |
| bill_length_mm     | 2         |
| bill_depth_mm      | 2         |
| flipper_length_mm  | 2         |
| body_mass_g        | 2         |
| sex                | 11        |
| year               | 0         |
└────────────────────┴───────────┘
```

В главе 4 мы обсудили различные стратегии обработки пропущенных значений.

# Чтение файлов с кодировкой, отличной от UTF-8

Каждый текстовый файл обладает своей *кодировкой* (character encoding). Кодировка представляет собой систему присвоения уникальных кодов отдельным символам, что позволяет обрабатывать текстовые данные на компьютере.

Polars по умолчанию считает, что все файлы CSV используют одну из самых распространенных кодировок *UTF-8*. Эта кодировка позволяет представить практически любой символ в стандарте Unicode, что дает возможность оперировать богатым набором символов в текстах на самых разных языках.

Если вы попытаетесь прочитать файл CSV в кодировке, отличной от UTF-8, вы в идеале получите ошибку[1]. Пример чтения такого файла:

```
pl.read_csv("data/directors.csv")
```

Вывод:

```
ComputeError: invalid utf-8 sequence
```

Очевидно, в файле *data/directors.csv* используется кодировка, отличная от UTF-8.

Можно, конечно, попытаться угадать кодировку в файле, но вы не всегда сможете с уверенностью сказать, что конкретная кодировка была распознана корректно.

Допустим, вам сказали, что в файле с данными содержатся имена директоров, среди которых есть представители азиатской культуры. Вы могли бы

[1] Мы сказали «в идеале», поскольку это позволит вам понять, что вы не указали нужную кодировку явным образом.

попробовать использовать самую распространенную кодировку для китайских символов, как показано ниже:

```
pl.read_csv("data/directors.csv", encoding="EUC-CN")
```

Вывод:

```
shape: (4, 3)
┌──────────┬──────┬──────────┐
│ name     ┆ born ┆ country  │
│ ---      ┆ ---  ┆ ---      │
│ str      ┆ i64  ┆ str      │
╞══════════╪══════╪══════════╡
│ 考侯     ┆ 1930 ┆ 泣塑     │
│ Verhoeven ┆ 1938 ┆ オランダ │
│ 弟宏     ┆ 1942 ┆ 泣塑     │
│ Tarantino ┆ 1963 ┆ 势柜     │
└──────────┴──────┴──────────┘
```

Вроде сработало. Или нет? При попытке перевода первой страны в столбце `country` с китайского на английский мы получим «Weeping plastic». Какое-то странное название для страны...

Вместо того чтобы пытаться наугад определить кодировку файла, можно воспользоваться помощью пакета *chardet*. Следующая функция возвращает кодировку для заданного файла. Давайте применим ее к нашему файлу CSV:

```
import chardet

def detect_encoding(filename: str) -> str:
    """Возвращает наиболее вероятную кодировку для файла."""

    with open(filename, "rb") as f:
        raw_data = f.read()
        result = chardet.detect(raw_data)
        return result["encoding"]

detect_encoding("data/directors.csv")
```

Вывод:

```
'EUC-JP'
```

Как видите, пакет chardet обнаружил другую кодировку в файле, – ту, которая чаще используется для японских символов. Давайте попробуем ей воспользоваться:

```
pl.read_csv("data/directors.csv", encoding="EUC-JP")
```

Вывод:

```
shape: (4, 3)
┌──────────┬──────┬──────────┐
│ name     ┆ born ┆ country  │
│ ---      ┆ ---  ┆ ---      │
```

```
┌────────────┬──────┬──────────┐
│ str        ┆ i64  ┆ str      │
╞════════════╪══════╪══════════╡
│ 深作       ┆ 1930 ┆ 日本     │
│ Verhoeven  ┆ 1938 ┆ オランダ │
│ 宮崎       ┆ 1942 ┆ 日本     │
│ Tarantino  ┆ 1963 ┆ 米国     │
└────────────┴──────┴──────────┘
```

Теперь все правильно. Доверьтесь нам, мы все проверили, честно!

Как видите, лучше не гадать относительно кодировки того или иного файла, а попытаться определить ее наверняка. И это справедливо не только для файлов CSV, но и для всех форматов на основе текста, таких как JSON, XML и HTML.

# Чтение данных из Excel

Хотя файлы CSV действительно очень популярны в среде аналитиков, таблицы Excel никуда не делись и по-прежнему широко используются в источниках, требующих ручного ввода информации или подразумевающих выполнение базового анализа данных.

Вы знаете, что в файлах Excel часто содержатся не только данные, как в CSV, но и сложная разметка, формулы, графики и т. д. Для бизнеса все эти украшательства могут представлять реальный интерес, но они явно не облегчают процесс импорта данных из таблиц Excel в Polars. Конечно, нам бы хотелось, чтобы в таблицах были только данные в чистом виде, без нагромождений.

Для чтения данных из Excel в датафреймы Polars вы можете воспользоваться пакетами *calamine*, *xlsx2csv* или *openpyxl* (подробная инструкция по установке пакетов находится в главе 2). По умолчанию в Polars используется пакет calamine, написанный на Rust, который позволяет очень быстро и эффективно сериализовать и десериализовать электронные таблицы. Что касается пакета xlsx2csv, то он конвертирует файлы Excel в формат CSV в памяти при помощи функции `pl.read_csv()`. Библиотека openpyxl значительно уступает по скорости первым двум, но зато способна автоматически приводить типы данных, что делает ее полезной в случаях, когда пакеты calamine и xlsx2csv не позволяют импортировать какой-то сложный файл Excel. В наших примерах мы будем использовать пакет calamine.

Давайте прочитаем данные из файла *data/top-2000-2023.xlsx*, в котором хранится плейлист ежегодной нидерландской радиопрограммы Top2000. В файле содержится информация о 2000 самых популярных песен по итогам опроса слушателей в 2023 году.

```
songs = pl.read_excel("data/top2000-2023.xlsx") ❶
songs
```

❶ Имена столбцов написаны на нидерландском языке и соответствуют позиции в списке, названию песни, исполнителю и году (любопытный факт: после фризского языка нидерландский является ближайшим родственником английского).

Вывод:

```
shape: (2_000, 4)
┌──────────┬──────────────────────┬──────────────────┬──────┐
│ positie  ┆ titel                ┆ artiest          ┆ jaar │
│ ---      ┆ ---                  ┆ ---              ┆ ---  │
│ i64      ┆ str                  ┆ str              ┆ i64  │
╞══════════╪══════════════════════╪══════════════════╪══════╡
│ 1        ┆ Bohemian Rhapsody    ┆ Queen            ┆ 1975 │
│ 2        ┆ Roller Coaster       ┆ Danny Vera       ┆ 2019 │
│ 3        ┆ Hotel California     ┆ Eagles           ┆ 1977 │
│ 4        ┆ Piano Man            ┆ Billy Joel       ┆ 1974 │
│ 5        ┆ Fix You              ┆ Coldplay         ┆ 2005 │
│ …        ┆ …                    ┆ …                ┆ …    │
│ 1996     ┆ Charlie Brown        ┆ Coldplay         ┆ 2011 │
│ 1997     ┆ Beast Of Burden      ┆ Bette Midler     ┆ 1984 │
│ 1998     ┆ It Was A Very Good Y… ┆ Frank Sinatra    ┆ 1968 │
│ 1999     ┆ Hou Van Mij          ┆ 3JS              ┆ 2008 │
│ 2000     ┆ Drivers License      ┆ Olivia Rodrigo   ┆ 2021 │
└──────────┴──────────────────────┴──────────────────┴──────┘
```

В табл. 6.3 перечислены наиболее часто используемые аргументы функции `pl.read_excel()`.

***Таблица 6.3. Часто используемые аргументы функции*** *`pl.read_excel()`*

| Аргумент | Описание |
|---|---|
| `source` | Путь к файлу или объекту |
| `sheet_id` | Номер листа для импорта (0 – для всех листов). Если не задан этот параметр и параметр `sheet_name`, по умолчанию используется значение 1. При выборе нескольких листов функция вернет словарь с именами листов в качестве ключей и датафреймами в качестве значений |
| `sheet_name` | Имя листа для загрузки. Не может быть использован вместе с аргументом `sheet_id` |
| `engine` | Используемый для обработки файла Excel пакет. Может принимать значения calamine, xlsx2csv или openpyxl |
| `engine_options` | Дополнительные опции, передаваемые конструктору движка (не используется с движком calamine) |
| `read_options` | Дополнительные опции, передаваемые функции чтения файла движка (не используется с движком openpyxl) |
| `has_header` | Индикатор наличия заголовков в файле |

Если вы обнаружите, что функция *`pl.read_excel()`* не справляется с загрузкой ваших данных, мы рекомендуем воспользоваться движком openpyxl или xlsx2csv с указанием дополнительных опций в аргументе `engine_options`. За подробностями вы можете обратиться к официальной документации функции.

# Работа с несколькими файлами

Если ваши данные рассредоточены по разным файлам, имеющим одинаковый формат и схему, вы можете прочитать их в один заход.

Допустим, нам необходимо собрать данные об акциях трех компаний: ASML Holding N.V. (ASML), NVIDIA Corporation (NVDA) и Taiwan Semiconductor Manufacturing Company Limited (TSM). При этом данные находятся в разных файлах CSV (по одному для каждой компании и года). Файлы именуются в соответствии со следующим шаблоном: *data/stock/<тикер>/<год>.csv*. Например: *data/stock/nvda/2010.csv* и *data/stock/asml/2022.csv*.

Поскольку все наши файлы имеют один формат и схему, мы можем воспользоваться маской, которая может включать следующие специальные символы:

- звездочка (`*`) соответствует любому количеству символов в строке (включая ноль). К примеру, шаблон `*.csv` соответствует всем файлам, заканчивающимся на *.csv*;
- вопросительный знак (`?`) указывает на наличие ровно одного символа. Например, шаблон `file?.csv` соответствует файлу *file1.csv* или *fileA.csv*, но не файлу *file12.csv*;
- квадратные скобки (`[]`) указывают на наличие любого символа из набора. Например, шаблон `file-[ab].csv` соответствует файлам *file-a.csv* и *file-b.csv*, но не *file-c.csv*. А шаблон `file-[0-9].csv` соответствует файлам *file-0.csv*, *file-1.csv*, *file-2.csv* и до *file-9.csv*.

Таким образом, для загрузки информации по компании NVIDIA за период с 2010 по 2019 год можно воспользоваться следующим шаблоном:

```
pl.read_csv("data/stock/nvda/201?.csv")
```

Вывод:

```
shape: (2_516, 8)
```

| symbol | date | open | … | close | adj close | volume |
|---|---|---|---|---|---|---|
| str | str | f64 | | f64 | f64 | i64 |
| NVDA | 2010-01-04 | 4.6275 | … | 4.6225 | 4.240429 | 80020400 |
| NVDA | 2010-01-05 | 4.605 | … | 4.69 | 4.30235 | 72864800 |
| NVDA | 2010-01-06 | 4.6875 | … | 4.72 | 4.32987 | 64916800 |
| NVDA | 2010-01-07 | 4.695 | … | 4.6275 | 4.245015 | 54779200 |
| NVDA | 2010-01-08 | 4.59 | … | 4.6375 | 4.254189 | 47816800 |
| … | … | … | … | … | … | … |
| NVDA | 2019-12-24 | 59.549999 | … | 59.654999 | 59.422798 | 13886400 |
| NVDA | 2019-12-26 | 59.689999 | … | 59.797501 | 59.564739 | 18285200 |
| NVDA | 2019-12-27 | 59.950001 | … | 59.217499 | 58.987 | 25464400 |

```
| NVDA   | 2019-12-30 | 58.997501 | … | 58.080002 | 57.853928 | 25805600 |
| NVDA   | 2019-12-31 | 57.724998 | … | 58.825001 | 58.596027 | 23100400 |
└────────┴────────────┴───────────┴───┴───────────┴───────────┴──────────┘
```

Для чтения всех файлов из директории *data/stock* необходимо воспользоваться шаблоном с двумя звездочками, поскольку файлы располагаются в разных папках:

```
all_stocks = pl.read_csv("data/stock/**/*.csv")
all_stocks
```

Вывод:

```
shape: (18_476, 8)
┌────────┬────────────┬──────────┬───┬──────────┬───────────┬──────────┐
| symbol | date       | open     | … | close    | adj close | volume   |
| ---    | ---        | ---      |   | ---      | ---       | ---      |
| str    | str        | f64      |   | f64      | f64       | i64      |
╞════════╪════════════╪══════════╪═══╪══════════╪═══════════╪══════════╡
| ASML   | 1999-01-04 | 11.765625 | … | 12.140625 | 7.522523  | 1801867  |
| ASML   | 1999-01-05 | 11.859375 | … | 13.96875  | 8.655257  | 8241600  |
| ASML   | 1999-01-06 | 14.25     | … | 16.875    | 10.456018 | 16400267 |
| ASML   | 1999-01-07 | 14.742188 | … | 16.851563 | 10.441495 | 17722133 |
| ASML   | 1999-01-08 | 16.078125 | … | 15.796875 | 9.787995  | 10696000 |
| …      | …          | …        | … | …         | …         | …        |
| TSM    | 2023-06-26 | 102.019997 | … | 100.110001 | 99.125954 | 8560000  |
| TSM    | 2023-06-27 | 101.150002 | … | 102.080002 | 101.076591 | 9732000  |
| TSM    | 2023-06-28 | 100.5    | … | 100.919998 | 99.927986 | 8160900  |
| TSM    | 2023-06-29 | 101.339996 | … | 100.639999 | 99.650742 | 7383900  |
| TSM    | 2023-06-30 | 101.400002 | … | 100.919998 | 99.927986 | 11701700 |
└────────┴────────────┴──────────┴───┴──────────┴───────────┴──────────┘
```

Если вам не удается описать все требуемые файлы для загрузки с помощью маски, вы можете прибегнуть к ручному методу отбора. Для этого выполните следующие действия.

1. Создайте список необходимых вам имен файлов.
2. Прочитайте эти файлы с помощью подходящей функции Polars (например, `pl.read_csv()`).
3. Объедините датафреймы в Polars при помощи функции *`pl.concat()`*.

Ниже приведен пример чтения данных по акциям компании ASML только за високосные годы:

```
import calendar

filenames = [
    f"data/stock/asml/{year}.csv"
    for year in range(1999, 2024)
    if calendar.isleap(year)
]

filenames
```

Вывод:

```
['data/stock/asml/2000.csv',
'data/stock/asml/2004.csv',
'data/stock/asml/2008.csv',
'data/stock/asml/2012.csv',
'data/stock/asml/2016.csv',
'data/stock/asml/2020.csv']
```

```
pl.concat(pl.read_csv(f) for f in filenames)
```

Вывод:

```
shape: (1_512, 8)
```

| symbol | date | open | … | close | adj close | volume |
|---|---|---|---|---|---|---|
| str | str | f64 | | f64 | f64 | i64 |
| ASML | 2000-01-03 | 43.875 | … | 43.640625 | 27.040424 | 1121600 |
| ASML | 2000-01-04 | 41.953125 | … | 40.734375 | 25.239666 | 968800 |
| ASML | 2000-01-05 | 39.28125 | … | 39.609375 | 24.542597 | 1458133 |
| ASML | 2000-01-06 | 36.75 | … | 37.171875 | 23.032274 | 3517867 |
| ASML | 2000-01-07 | 36.867188 | … | 38.015625 | 23.555077 | 1631200 |
| … | … | … | … | … | … | … |
| ASML | 2020-12-24 | 478.950012 | … | 483.089996 | 468.836365 | 271900 |
| ASML | 2020-12-28 | 487.140015 | … | 480.23999 | 466.070496 | 449300 |
| ASML | 2020-12-29 | 489.450012 | … | 484.01001 | 469.729218 | 377200 |
| ASML | 2020-12-30 | 488.130005 | … | 489.910004 | 475.455231 | 381900 |
| ASML | 2020-12-31 | 490.0 | … | 487.720001 | 473.329803 | 312700 |

Подробнее о функции `pl.concat()` мы поговорим в главе 14.

# Чтение из файлов Parquet

*Parquet* представляет собой формат колоночного хранения данных, оптимизированный для использования во фреймворках, предназначенных для работы с большими данными, таких как Apache Spark, DuckDB и, конечно, Polars. Хотя формат Parquet разработан для эффективного хранения данных на диске, он успешно дополняет формат хранения в памяти Apache Arrow, лежащий в основе Polars. Это позволяет легко переходить от хранения на диске к хранению в памяти, о чем мы подробно поговорим в главе 18.

В сравнении со строчными форматами, такими как CSV и Excel, Parquet показывает наибольшую эффективность при чтении и записи больших наборов данных, особенно если вам необходимо обработать конкретные столбцы. Кроме того, Parquet поддерживает сложные вложенные структуры данных, в то время как CSV и Excel представляют собой плоские форматы. Это делает

его более подходящим для работы со сложными наборами данных. Также формат Parquet включает в себя схему данных, что позволяет избежать многих ошибок, присущих чтению файлов CSV.

Ниже приведен пример загрузки данных о поездках знаменитых желтых такси в Нью-Йорке:

```
%%time
trips = pl.read_parquet("data/taxi/yellow_tripdata_*.parquet")
trips
```

Вывод:

```
CPU times: user 2 µs, sys: 0 ns, total: 2 µs
Wall time: 4.05 µs
shape: (39_656_098, 19)
```

| VendorID | tpep_pickup_datetime | … | congestion_surcharge | airport_fee |
|---|---|---|---|---|
| i64 | datetime[ns] | | f64 | f64 |
| 1 | 2022-01-01 00:35:40 | … | 2.5 | 0.0 |
| 1 | 2022-01-01 00:33:43 | … | 0.0 | 0.0 |
| 2 | 2022-01-01 00:53:21 | … | 0.0 | 0.0 |
| 2 | 2022-01-01 00:25:21 | … | 2.5 | 0.0 |
| 2 | 2022-01-01 00:36:48 | … | 2.5 | 0.0 |
| … | … | … | … | … |
| 2 | 2022-12-31 23:46:00 | … | null | null |
| 2 | 2022-12-31 23:13:24 | … | null | null |
| 2 | 2022-12-31 23:00:49 | … | null | null |
| 1 | 2022-12-31 23:02:50 | … | null | null |
| 2 | 2022-12-31 23:00:15 | … | null | null |

На средненьком ноутбуке операция чтения порядка 40 млн записей при помощи функции *pl.read_parquet()* заняла всего несколько мкс. В Jupyter вы можете замерить время выполнения ячейки с помощью магической команды *%time*.

В табл. 6.4 приведен список наиболее часто используемых аргументов функции `pl.read_parquet()`.

***Таблица 6.4. Часто используемые аргументы функции*** *pl.read_parquet()*

| Аргумент | Описание |
|---|---|
| `Source` | Путь к файлу или объекту. Если указать директорию, будут прочитаны все файлы из нее |
| `Columns` | Столбцы для отбора. Принимает список индексов столбцов (начиная с нуля) или имен столбцов |
| `n_rows` | Указывает на необходимость прекратить операцию чтения после обработки заданного количества строк. Используется только с аргументом `use_pyarrow=False` |
| `use_pyarrow` | Флаг использования движка PyArrow вместо движка чтения Parquet, родного для Rust. Движок PyArrow отличается повышенной стабильностью. Значение по умолчанию – `False` |

Скорость и надежность делают Parquet лучшим, по моему мнению, форматом для работы с датафреймами. На протяжении книги мы еще не раз поработаем с этим форматом.

# Чтение JSON и NDJSON

Этот раздел мы посвятим операциям чтения из форматов JavaScript Object Notation (JSON) и Newline Delimited JSON (NDJSON).

## JSON

*JSON* представляет собой текстовый формат, который легко воспринимается человеком и эффективно разбирается и генерируется с помощью автоматизированных инструментов. В отличие от CSV и Excel, формат JSON может содержать вложенные структуры данных. Такая гибкость обусловила выбор этого формата в качестве приоритетного в области работы с API, базами данных NoSQL и конфигурационными файлами.

Давайте взглянем на содержимое файла *data/pokedex.json*:

```
! cat data/pokedex.json
```

Вывод:

```
{
    "pokemon": [{
        "id": 1,
        "num": "001",
        "name": "Bulbasaur",
        "img": "http://www.serebii.net/pokemongo/pokemon/001.png",
        "type": [
            "Grass",
            "Poison"
        ],
        "height": "0.71 m",
        "weight": "6.9 kg",
        "candy": "Bulbasaur Candy",
        "candy_count": 25,
        "egg": "2 km",
        "spawn_chance": 0.69,
        "avg_spawns": 69,
        "spawn_time": "20:00",
        "multipliers": [1.58],
        "weaknesses": [
            "Fire",
            "Ice",
            "Flying",
```

```
            "Psychic"
        ],
        "next_evolution": [{
            "num": "002",
            "name": "Ivysaur"
        }, {
            "num": "003",
            "name": "Venusaur"
        }]
    }, {
    ... with 4053 more lines
```

Этот файл JSON начинается и заканчивается фигурными скобками – это означает, что весь файл представляет собой один объект JSON. Именно использование фигурных скобок позволяет формату JSON хранить вложенные структуры.

Объект имеет один ключ `pokemon`, содержащий целый список объектов. В первых 33 строках вывода мы видим лишь один из этих вложенных объектов – с именем `Bulbasaur`. Этот объект, в свою очередь, содержит несколько ключей со своими объектами в качестве значений. Такая гибкость имеет множество преимуществ, но, как вы увидите далее, она может осложнять процесс чтения файлов JSON в Polars.

Давайте посмотрим, что произойдет при попытке чтения файла JSON в датафрейм Polars при помощи функции *`pl.read_json()`*:

```
pokedex = pl.read_json("data/pokedex.json")
pokedex
```

Вывод:

```
shape: (1, 1)
┌──────────────────────────────────────────────────────────────────────────┐
│ pokemon                                                                  │
│ ---                                                                      │
│ list[struct[17]]                                                         │
╞══════════════════════════════════════════════════════════════════════════╡
│ [{1,"001","Bulbasaur","http://www.serebii.net/pokemongo/pokemon/001.png",["G │
│ rass", "Poison"],"0.71 m","6.9 kg","Bulbasaur Candy","2 km",0.69,69.0,"20:0… │
└──────────────────────────────────────────────────────────────────────────┘
```

Обратите внимание, что весь файл был помещен в одно значение. Причина в том, что наш файл JSON содержит только один ключ `pokemon`, который соответствует списку объектов. По умолчанию Polars не делает никаких предположений о том, как привести вложенную структуру к табличному виду.

К счастью, в Polars присутствуют два удобных метода для ручного выравнивания подобных структур данных. Это метод *`explode()`*, преобразующий каждый объект в списке в новую строку, и метод *`unnest()`*, превращающий каждый ключ объекта в новый столбец. Более детально мы обсудим эти методы в главе 12, а сейчас просто немного поработаем с нашим объектом `pokedex`:

```
(
    pokedex.explode("pokemon")
    .unnest("pokemon")
    .select("id", "name", "type", "height", "weight")
)
```

Вывод:

```
shape: (151, 5)
┌─────┬────────────┬──────────────────────┬────────┬────────┐
│ id  ┆ name       ┆ type                 ┆ height ┆ weight │
│ --- ┆ ---        ┆ ---                  ┆ ---    ┆ ---    │
│ i64 ┆ str        ┆ list[str]            ┆ str    ┆ str    │
╞═════╪════════════╪══════════════════════╪════════╪════════╡
│ 1   ┆ Bulbasaur  ┆ ["Grass", "Poison"]  ┆ 0.71 m ┆ 6.9 kg │
│ 2   ┆ Ivysaur    ┆ ["Grass", "Poison"]  ┆ 0.99 m ┆ 13.0 kg│
│ 3   ┆ Venusaur   ┆ ["Grass", "Poison"]  ┆ 2.01 m ┆ 100.0 kg│
│ 4   ┆ Charmander ┆ ["Fire"]             ┆ 0.61 m ┆ 8.5 kg │
│ 5   ┆ Charmeleon ┆ ["Fire"]             ┆ 1.09 m ┆ 19.0 kg│
│ …   ┆ …          ┆ …                    ┆ …      ┆ …      │
│ 147 ┆ Dratini    ┆ ["Dragon"]           ┆ 1.80 m ┆ 3.3 kg │
│ 148 ┆ Dragonair  ┆ ["Dragon"]           ┆ 3.99 m ┆ 16.5 kg│
│ 149 ┆ Dragonite  ┆ ["Dragon", "Flying"] ┆ 2.21 m ┆ 210.0 kg│
│ 150 ┆ Mewtwo     ┆ ["Psychic"]          ┆ 2.01 m ┆ 122.0 kg│
│ 151 ┆ Mew        ┆ ["Psychic"]          ┆ 0.41 m ┆ 4.0 kg │
└─────┴────────────┴──────────────────────┴────────┴────────┘
```

В табл. 6.5 приведены часто используемые аргументы функций `pl.read_json()` и `pl.read_ndjson()`, применяющихся для чтения форматов JSON и NDJSON, о котором мы поговорим далее.

***Таблица 6.5. Часто используемые аргументы функций*** *`pl.read_json()`* ***и*** *`pl.read_ndjson()`*

| Аргумент | Описание |
|---|---|
| `Source` | Путь к файлу или объекту |
| `Schema` | Схему датафрейма можно объявить разными способами.<br>1. В виде словаря с парами `{имя: тип}`. Если тип `None`, он будет выведен автоматически.<br>2. В виде списка имен столбцов. В этом случае типы выводятся автоматически.<br>3. В виде списка пар `(имя, тип)`. Этот способ аналогичен способу со словарем |
| `schema_overrides` | Поддерживает спецификацию типов или переопределяет один либо несколько столбцов. Любые типы, выведенные из аргумента `schema`, будут переопределены |

# NDJSON

*NDJSON* представляет собой удобный формат хранения и потоковой передачи структурированных данных, которые должны обрабатываться по одной

записи за раз. По сути, речь идет о коллекции объектов JSON, разделенных символами переноса строки.

Каждая строка в наборе данных NDJSON представлена отдельным объектом JSON, но весь файл в целом нельзя рассматривать как валидный массив JSON, поскольку символы переноса строки не входят в базовый синтаксис JSON. Преимущества этого формата заключаются в легкости добавления и чтения данных построчно, что может быть полезно в сценариях, связанных с потоковой передачей, или при работе с большими данными, не помещающимися целиком в памяти. Формат NDJSON, в частности, удобно использовать при работе с логами RESTful API.

Мы заранее подготовили файл *data/wikimedia.ndjson* на основе потока из Wikimedia API, предварительно немного очистив его. Ниже показаны первые пять строк содержимого файла:

```
! cat data/wikimedia.ndjson
```

Вывод:

```
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia....
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia....
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia....
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia....
{"$schema":"/mediawiki/recentchange/1.0.0","meta":{"uri":"https://en.wikipedia....
... with 95 more lines
```

Как видите, на каждой строке располагается отдельный объект JSON. Давайте внимательнее присмотримся к первому из них:

```
from json import loads
from pprint import pprint

with open("data/wikimedia.ndjson") as f:
    pprint(loads(f.readline()))
```

Вывод:

```
{'$schema': '/mediawiki/recentchange/1.0.0',
 'bot': False,
 'comment': '/* League champions, runners-up and play-off finalists */',
 'id': 1659529639,
 'length': {'new': 91166, 'old': 91108},
 'meta': {'domain': 'en.wikipedia.org',
          'dt': '2023-07-29T07:51:39Z',
          'id': '0416300b-980c-45bb-b0a2-c9d7a9e2b7eb',
          'offset': 4820784717,
          'partition': 0,
          'request_id': 'ea0541fb-4e72-4fc3-82f0-6c26651b2043',
          'stream': 'mediawiki.recentchange',
          'topic': 'eqiad.mediawiki.recentchange',
          'uri': 'https://en.wikipedia.org/wiki/EFL_Championship'},
 'minor': False,
 'namespace': 0,
```

```
 'notify_url': 'https://en.wikipedia.org/w/index.php?diff=1167689309&oldid=1166...
 'parsedcomment': '<span dir="auto"><span class="autocomment"><a '
                  'href="/wiki/EFL_Championship#League_champions,_runners-up_an...
                  'title="EFL Championship">→\u200eLeague champions, '
                  'runners-up and play-off finalists</a></span></span>',
 'revision': {'new': 1167689309, 'old': 1166824248},
 'server_name': 'en.wikipedia.org',
 'server_script_path': '/w',
 'server_url': 'https://en.wikipedia.org',
 'timestamp': 1690617099,
 'title': 'EFL Championship',
 'title_url': 'https://en.wikipedia.org/wiki/EFL_Championship',
 'type': 'edit',
 'user': '87.12.215.232',
 'wiki': 'enwiki'}
```

Обратите внимание, что этот объект JSON обладает вложенной структурой. Три ключа, а именно `length`, `meta` и `revision`, включают в себя несколько ключей и значений. Посмотрим, как Polars справится с такими данными при помощи функции *`pl.read_ndjson()`*:

```
wikimedia = pl.read_ndjson("data/wikimedia.ndjson")
wikimedia
```

Вывод:

```
shape: (100, 20)
```

| $schema | meta | … | wiki | parsedcomment |
|---|---|---|---|---|
| str | struct[9] | | str | str |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | <… |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | <… |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | Nominated for dele… |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | Rescuing 1 sources… |
| … | … | … | … | … |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | <… |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | Ce |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | |
| /mediawiki/recentc… | {"https://en.wikip… | … | enwiki | <… |

Как и в случае с объектом `pokedex`, мы можем воспользоваться методом `unnest()` для трансформации ключей в отдельные столбцы:

```
(
    wikimedia.rename({"id": "edit_id"})
    .unnest("meta")
    .select("timestamp", "title", "user", "comment")
)
```

Вывод:

```
shape: (100, 4)
```

| timestamp | title | user | comment |
|---|---|---|---|
| i64 | str | str | str |
| 1690617099 | EFL Championship | 87.12.215.232 | /* League champio… |
| 1690617102 | Lim Sang-choon | Preferwiki | |
| 1690617104 | Higher | Ss112 | /* Albums */ add |
| 1690617104 | International Pok… | Piotrus | Nominated for del… |
| 1690617105 | Abdul Hamid Khan … | InternetArchiveBo… | Rescuing 1 source… |
| … | … | … | … |
| 1690617238 | Havering Resident… | MRSC | /* 2018 election … |
| 1690617235 | Olha Kharlan | 2603:7000:2101:AA… | Ce |
| 1690617238 | Mukim Kota Batu | Pangalau | |
| 1690617239 | User:IDK1213safas… | 94.101.29.27 | |
| 1690617234 | List of bus route… | Pedroperezhumbert… | /* Non-TfL bus ro… |

Заметьте, что нам пришлось переименовать столбец `id` в `edit_id`. В противном случае метод `unnest()` завершился бы неудачей по причине наличия дублирующихся имен столбцов. Polars требует, чтобы все столбцы в датафрейме имели уникальные имена – это позволяет более эффективно применять выражения. В связи с этим иногда приходится предварительно переименовывать элементы структур, которые должны быть преобразованы в колонки. В данном случае мы разворачиваем в столбцы поле `meta`, содержащее поле `id`, тогда как в нашем датафрейме уже присутствует столбец с таким именем.

# Другие форматы файлов

Polars также поддерживает форматы Arrow IPC (Feather версии 2), Apache Avro, таблицы Delta Lake и наборы данных PyArrow. Для чтения из этих форматов в библиотеке предусмотрены функции *`pl.read_ipc()`*, *`pl.read_avro()`*, *`pl.read_delta()`* и *`pl.scan_pyarrow_dataset()`* соответственно.

Если вам необходимо обработать в Polars формат, не поддерживаемый этой библиотекой, на помощь может прийти pandas. Библиотеке pandas уже больше 14 лет, так что неудивительно, что она способна работать с гораздо большим количеством форматов данных. Вы всегда можете преобразовать датафрейм pandas в датафрейм Polars с помощью функции *`pl.from_pandas()`*. Ниже приведен пример чтения со страницы HTML:

```
import pandas as pd

url = "https://en.wikipedia.org/wiki/List_of_Latin_abbreviations"
чpl.from_pandas(pd.read_html(url)[0]) ❶
```

❶ Поскольку страница содержит несколько таблиц, нам необходимо выбрать из них нужную.

Вывод:

```
shape: (63, 4)
```

| abbreviation | Latin | translation | usage and notes |
|---|---|---|---|
| str | str | str | str |
| AD | anno Domini | "in the year of t… | Used to label or … |
| a.i. | ad interim | "temporarily" | Used in business … |
| a.m. | ante meridiem | "before midday"[1… | Used on the twelv… |
| ca. c. | circa | "around", "about… | Used with dates t… |
| Cap. | capitulus | "chapter" | Used before a cha… |
| … | … | … | … |
| SOS | si opus sit | "if there is need… | A prescription in… |
| sic | sic | "thus" | Used when quoting… |
| stat. | statim | "immediately" | Often used in med… |
| viz. | videlicet | "namely", "to wit… | In contradistinct… |
| vs. v. | versus | "against" | Sometimes is not … |

Помимо HTML, pandas (не Polars) поддерживает чтение из текстовых файлов с полями фиксированного размера, HDF5, ORC, SAS, SPSS, Stata, XML и многих других форматов. Некоторые из этих форматов требуют установки вспомогательных пакетов. К примеру, предыдущий пример с чтением из файла HTML потребовал установки пакета lxml. Подробнее о поддержке разных форматов в pandas можно почитать в официальной документации по адресу *https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html*.

# Запросы к базам данных

Polars обеспечивает удобную работу с реляционными базами данных посредством универсальной функции *`pl.read_database()`*. Эта функция позволяет выполнять запросы SQL напрямую и извлекать результаты в виде датафрейма. Библиотека Polars поддерживает работу с большим количеством РСУБД, включая Postgres, MS SQL, MySQL, Oracle, SQLite и BigQuery.

На вход функция `pl.read_database()` принимает запрос на языке SQL и строку подключения к базе данных. С помощью строки подключения вы можете указать тип базы данных, ее расположение и, при необходимости, учетные данные. К примеру, строка для подключения к базе данных Postgres соответствует следующему шаблону: *postgres://логин:пароль@сервер:порт/база_данных*.

Обычно база данных располагается на отдельном сервере (или по крайней мере в отдельном процессе) и требует ввода учетных данных. В то же время

база данных SQLite представляет собой отдельный файл. Для простоты демонстрацию работы Polars с базами данных мы проведем именно на примере базы SQLite. Процедура работы с другими базами данных ничем не будет отличаться, за исключением передачи другой строки подключения и, при необходимости, использования особого диалекта SQL.

Мы для примера воспользуемся простой базой данных Sakila, разработанной командой MySQL и портированной в SQLite Брэдли Грантом (Bradley Grant). Со схемой данных в этой базе можно ознакомиться по адресу *https://github.com/bradleygrant/sakila-sqlite3*.

```
pl.read_database_uri(
    query="""
    SELECT
        f.film_id,
        f.title,
        c.name AS category,
        f.rating,
        f.length / 60.0 AS length
    FROM
        film AS f,
        film_category AS fc,
        category AS c
    WHERE
        fc.film_id = f.film_id
        AND fc.category_id = c.category_id
    LIMIT 10
    """,
    uri="sqlite:::data/sakila.db",
)
```

Вывод:

```
shape: (10, 5)
```

| film_id | title | category | rating | length |
|---|---|---|---|---|
| i64 | str | str | str | f64 |
| 1 | ACADEMY DINOSAUR | Documentary | PG | 1.433333 |
| 2 | ACE GOLDFINGER | Horror | G | 0.8 |
| 3 | ADAPTATION HOLES | Documentary | NC-17 | 0.833333 |
| 4 | AFFAIR PREJUDICE | Horror | G | 1.95 |
| 5 | AFRICAN EGG | Family | G | 2.166667 |
| 6 | AGENT TRUMAN | Foreign | PG | 2.816667 |
| 7 | AIRPLANE SIERRA | Comedy | PG-13 | 1.033333 |
| 8 | AIRPORT POLLOCK | Horror | R | 0.9 |
| 9 | ALABAMA DEVIL | Horror | PG-13 | 1.9 |
| 10 | ALADDIN CALENDAR | Sports | NC-17 | 1.05 |

Если SQL – не ваш конек, но вам нужно проверить успешность подключения к базе данных, вы можете воспользоваться простейшей конструкцией `SELECT * FROM table`, которая позволит вам извлечь все записи из заданных таблиц, а объединение и остальные работы выполнить в Polars. Например, следующие три запроса с дальнейшей обработкой в Polars позволят вам получить такой же результат, что и ранее:

```
db = "sqlite:::data/sakila.db"
films = pl.read_database_uri("SELECT * FROM film", db)
film_categories = pl.read_database_uri("SELECT * FROM film_category", db)
categories = pl.read_database_uri("SELECT * FROM category", db)

(
    films.join(film_categories, on="film_id", suffix="_fc")
    .join(categories, on="category_id", suffix="_c")
    .select(
        "film_id",
        "title",
        pl.col("name").alias("category"),
        "rating",
        pl.col("length") / 60,
    )
    .limit(10)
)
```

Вывод:

```
shape: (10, 5)
┌─────────┬──────────────────┬─────────────┬────────┬──────────┐
│ film_id ┆ title            ┆ category    ┆ rating ┆ length   │
│ ---     ┆ ---              ┆ ---         ┆ ---    ┆ ---      │
│ i64     ┆ str              ┆ str         ┆ str    ┆ f64      │
╞═════════╪══════════════════╪═════════════╪════════╪══════════╡
│ 1       ┆ ACADEMY DINOSAUR ┆ Documentary ┆ PG     ┆ 1.433333 │
│ 2       ┆ ACE GOLDFINGER   ┆ Horror      ┆ G      ┆ 0.8      │
│ 3       ┆ ADAPTATION HOLES ┆ Documentary ┆ NC-17  ┆ 0.833333 │
│ 4       ┆ AFFAIR PREJUDICE ┆ Horror      ┆ G      ┆ 1.95     │
│ 5       ┆ AFRICAN EGG      ┆ Family      ┆ G      ┆ 2.166667 │
│ 6       ┆ AGENT TRUMAN     ┆ Foreign     ┆ PG     ┆ 2.816667 │
│ 7       ┆ AIRPLANE SIERRA  ┆ Comedy      ┆ PG-13  ┆ 1.033333 │
│ 8       ┆ AIRPORT POLLOCK  ┆ Horror      ┆ R      ┆ 0.9      │
│ 9       ┆ ALABAMA DEVIL    ┆ Horror      ┆ PG-13  ┆ 1.9      │
│ 10      ┆ ALADDIN CALENDAR ┆ Sports      ┆ NC-17  ┆ 1.05     │
└─────────┴──────────────────┴─────────────┴────────┴──────────┘
```

Использование этого подхода приведет к передаче большого количества данных. Более эффективным считается выполнять всю необходимую обработку на стороне базы данных, а на клиента передавать только нужные столбцы итогового набора.

# Запись данных

В Polars предусмотрено большое количество методов для записи данных в файл. Понимание всех нюансов разных форматов файлов поможет вам сделать осознанный выбор в пользу того или иного варианта хранения данных.

## Запись в формате CSV

Одним из наиболее часто используемых форматов хранения данных является CSV. Этот формат поддерживается подавляющим большинством приложений, что делает его универсальным выбором во многих ситуациях. Для сохранения данных в этом формате необходимо воспользоваться методом *write_csv()*, как показано ниже:

```
all_stocks.write_csv("data/all_stocks.csv")
```

В табл. 6.6 перечислены наиболее часто используемые аргументы этого метода.

***Таблица 6.6. Часто используемые аргументы метода*** *write_csv()*

| Аргумент | Описание |
|---|---|
| File | Путь к файлу для записи. Если передать значение None (по умолчанию), результат будет выведен в виде строки |
| include_header | Индикатор включения заголовков (значение по умолчанию – True) |
| Separator | Символ разделителя (по умолчанию запятая) |
| Quote | Символ для обрамления строковых данных (по умолчанию двойная кавычка) |
| null_value | Строка для представления пропущенных значений (по умолчанию пустая строка) |

Поскольку в основе CSV лежит текстовый формат, его довольно легко читать. В то же время, как вы уже видели, использование этого формата может быть сопряжено с определенными сложностями в отношении кодировки, пропущенных значений и использования схемы.

## Запись в формате Excel

Если вам необходимо сохранить датафрейм в формате, который сможет открыть у себя на компьютере практически любой пользователь, оптимальным выбором будет Excel. Для сохранения данных в этом формате необходимо воспользоваться методом датафрейма *write_excel()*, как показано ниже:

```
all_stocks.write_excel("data/all_stocks.xlsx")
```

В табл. 6.7 перечислены наиболее часто используемые аргументы этого метода.

***Таблица 6.7. Часто используемые аргументы метода*** *`write_excel()`*

| Аргумент | Описание |
|---|---|
| `worksheet` | Имя листа для записи (по умолчанию `Sheet1`) |
| `Position` | Расположение таблицы на листе в нотации Excel (например, `A1`) или в виде кортежа из номера строки и столбца |
| `table_style` | Именованный стиль таблицы Excel (например, `Table Style Medium 4`) или словарь с одним или несколькими ключами из следующего списка: `style`, `first_column`, `last_column`, `banded_columns`, `banded_rows` |
| `column_widths` | Словарь вида `{colname:int}` или целочисленное значение, которое устанавливает (или переопределяет) ширину столбцов в таблице в пикселях. При передаче целочисленного значения все столбцы таблицы будут одной ширины |

Одно из главных преимуществ формата Excel состоит в поддержке рабочих книг со множеством листов и возможности внедрения стилевых параметров и формул непосредственно в данные. В то же время этот формат является двоичным, что не позволяет просмотреть данные в файле без использования Excel. Кроме того, это не лучший вариант для хранения больших данных.

# Запись в формате Parquet

При работе с достаточно объемными датафреймами формат Parquet можно считать идеальным универсальным выбором для хранения данных. С помощью метода *`write_parquet()`* вы можете легко сохранить свои данные в колоночном формате, как показано ниже:

```
all_stocks.write_parquet("data/all_stocks.parquet")
```

В табл. 6.8 перечислены часто используемые аргументы метода `write_parquet()`.

***Таблица 6.8. Часто используемые аргументы метода*** *`write_parquet()`*

| Аргумент | Описание |
|---|---|
| `File` | Имя файла для сохранения данных |
| `compression` | Выберите вариант `zstd` для эффективного сжатия данных. Вариант `lz4` подойдет для быстрого сжатия и распаковывания данных. А вариант `snappy` предназначен для обратной совместимости с более старыми движками чтения Parquet |
| `compression_level` | Уровень сжатия данных. Чем выше уровень, тем меньше места на диске будет занимать итоговый файл. Каждый алгоритм располагает своим перечнем степеней сжатия. Обратитесь к документации для выбора наиболее подходящего способа компрессии данных |

Parquet – высокоэффективный формат данных, обеспечивающий высокую степень сжатия и поддерживающий вложенные структуры данных. Кроме того, он хранит информацию о схеме данных, что позволяет более последовательно и единообразно извлекать хранящуюся в файле информацию. В то же время формат Parquet не является столь же универсальным и распространенным, как CSV или Excel, что требует использования специализированных инструментов для его чтения.

## Другие варианты хранения

Библиотека Polars также поддерживает операцию записи в другие форматы, такие как Avro и JSON. При выборе оптимального формата для хранения данных необходимо учитывать такие факторы, как предназначение данных, совместимость с другими программными продуктами, размер набора данных и сложность используемых в них структур.

# Заключение

В этой главе вы узнали, что:

- библиотека Polars располагает богатым функционалом для чтения и записи данных в самых разных форматах, включая CSV, Excel, Parquet и JSON;
- в Polars поддерживается эффективное чтение/запись, ленивая загрузка и обработка сложных структур данных, например вложенных JSON или Parquet;
- дополнительные пакеты вроде calamine, chardet и ConnectorX помогают при выполнении задач, связанных с чтением данных из файлов Excel, распознаванием кодировок, подключением к базам данных и т. д.;
- для обработки пропущенных значений, декодирования файлов и указания специфических свойств, таких как разделитель в файлах CSV, вы можете воспользоваться соответствующими аргументами методов и функций;
- для обработки сразу нескольких файлов с данными можно воспользоваться шаблоном в виде маски, а для управления вложенными структурами – удобными методами `explode()` и `unnest()`;
- для работы с большими данными более предпочтительным форматом файлов будет Parquet, а не привычные CSV или Excel.

В следующей главе мы перейдем к одной из важнейших тем, связанных с библиотекой Polars, а именно к выражениям.

# ЧАСТЬ III

# Выражения

# Глава 7

# Введение в выражения

Цель этой главы состоит в знакомстве с выражениями в Polars, которым API этой библиотеки обязана своей гибкостью и мощью. Здесь мы заложим основы того, о чем будем подробно говорить на протяжении всей третьей части книги.

**Выражения в Polars и регулярные выражения**

Не стоит путать выражения, использующиеся в Polars, с регулярными выражениями (*regular expression*, или *regex*). Регулярные выражения представляют собой последовательность символов, используемую для выделения фрагментов из текста по определенному шаблону. К примеру, регулярное выражение `[Pp](ol|and)ar?s` позволит найти в тексте упоминания как pandas, так и Polars, но пропустит вхождения panda или polaris. В Polars присутствует несколько функций и методов, способных работать с регулярными выражениями, таких как `pl.col()` для выбора столбцов и `Expr.str.replace()` для замены значений. Чтобы глубже погрузиться в тему регулярных выражений, можно посетить сайт Гранта Скиннера (Grant Skinner) *https://regexr.com* или прочитать книгу *Introducing Regular Expressions* Майкла Фицджеральда (Michael Fitzgerald).

*Выражения* (expression) в Polars – это повторно используемые строительные блоки, позволяющие выполнять множество разных операций по работе с данными, включая выбор существующих столбцов, создание новых, фильтрацию строк по условию и вычисление агрегаций. В общем, выражения в Polars встречаются повсюду.

Выражения – это очень важная и объемная тема, в связи с чем мы посвятили ей целые три главы книги, как показано на рис. 7.1. Кроме того, в главе 13 мы обсудим разные методы, доступные посредством так называемых пространств имен (namespace), о которых мы поговорим в следующем разделе.

В этой главе вы узнаете:

- что такое выражения;
- где можно использовать выражения;
- как создавать выражения на основе существующих столбцов;
- как создавать выражения на основе литеральных значений;
- как создавать выражения на основе диапазонов;

- как переименовывать выражения;
- почему использование выражений является рекомендованной практикой работы в Polars.

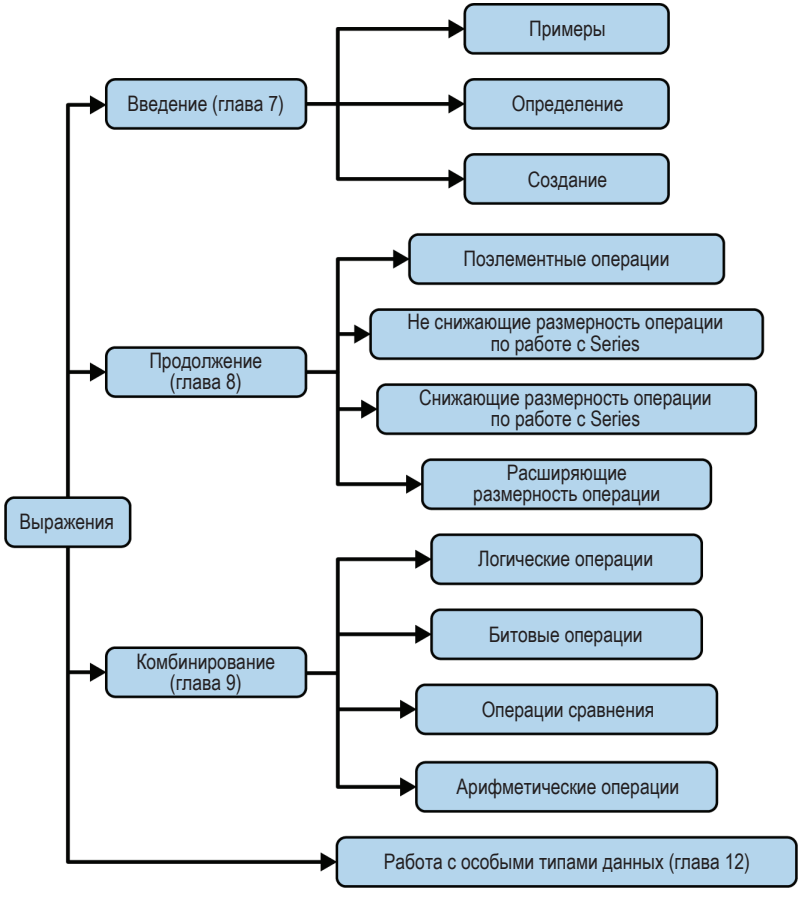



**Рис. 7.1** ❖ Методы выражений в Polars, разделенные на три главы книги

Все инструкции по загрузке и установке требуемых файлов вы найдете в главе 2. Мы будем исходить из предположения о том, что все рабочие файлы вы поместили в директорию *data*.

**Сначала примеры, затем определение**

Без погружения в контекст постичь смысл выражений может быть очень непросто. Именно поэтому перед формальным определением выражений и детальным описанием их работы мы решили показать вам несколько примеров их употребления. Это позволит вам более плавно войти в эту непростую тему.

В главах 8 и 9 вы узнаете, как можно расширять и комбинировать выражения.

# Методы и пространства имен

Класс *pl.Expr*, при помощи которого в Polars реализованы выражения, содержал на момент написания книги порядка 350 (!) методов. Более сотни методов для работы с выражениями доступны посредством так называемых *пространств имен* (namespace), представляющих собой группы методов для работы с конкретными типами данных.

К примеру, пространство имен *Expr.str* включает в себя методы для работы со строками, *Expr.dt* – для работы с календарными типами, а *Expr.cat* – с категориями. Эти типы и связанные с ними методы будут описаны в главе 12. Сейчас же мы сосредоточимся на более общих методах для работы с выражениями.

# Выражения в примерах

Выражения в Polars проявляют свою мощь в момент их применения. Это может звучать как-то банально, но в действительности выражения сами по себе ничего не делают. Они следуют ленивой парадигме, подобно ленивым датафреймам, которые мы с вами обсуждали ранее.

Перед погружением в подробности работы выражений мы рассмотрим несколько примеров:

- выбор столбцов с помощью метода `select()`;
- создание новых столбцов с помощью метода `with_columns()`;
- фильтрация строк с помощью метода `filter()`;
- агрегация с помощью метода `group_by()`;
- сортировка с помощью метода `sort()`.

Разбирая приведенные примеры, обращайте пристальное внимание не на сами методы, а на использование в них выражений. С каждым методом в отдельности мы познакомимся в соответствующих главах и разделах.

В примерах мы будем использовать простой набор данных из десяти фруктов[1]:

```
fruit = pl.read_csv("data/fruit.csv")
fruit
```

Вывод:

```
shape: (10, 5)
┌─────────────┬────────┬────────┬──────────┬──────────────┐
│ name        ┆ weight ┆ color  ┆ is_round ┆ origin       │
│ ---         ┆ ---    ┆ ---    ┆ ---      ┆ ---          │
```

[1] Да, авокадо действительно относится к фруктам – если быть точным, это ягода с одной косточкой.

```
| str         | i64  | str    | bool  | str           |
╞═════════════╪══════╪════════╪═══════╪═══════════════╡
| Avocado     | 200  | green  | false | South America |
| Banana      | 120  | yellow | false | Asia          |
| Blueberry   | 1    | blue   | false | North America |
| Cantaloupe  | 2500 | orange | true  | Africa        |
| Cranberry   | 2    | red    | false | North America |
| Elderberry  | 1    | black  | false | Europe        |
| Orange      | 130  | orange | true  | Asia          |
| Papaya      | 1000 | orange | false | South America |
| Peach       | 150  | orange | true  | Asia          |
| Watermelon  | 5000 | green  | true  | Africa        |
└─────────────┴──────┴────────┴───────┴───────────────┘
```

Методы, которые мы продемонстрируем, доступны также и для ленивых датафреймов, но поскольку мы имеем дело всего с десятью строками, традиционного датафрейма нам хватит за глаза. Кроме того, в отсутствие необходимости применять метод `collect()` наши примеры будут короче.

## Выбор столбцов при помощи выражений

Вы можете выбрать один или несколько существующих столбцов в датафрейме с помощью метода *`select()`*. При этом столбцы, не упомянутые в аргументах, будут исключены из вывода. Кроме того, в результате применения любых выражений в методе `select()`, подразумевающих создание нового столбца, в результирующем датафрейме появятся новые колонки. С помощью приведенного ниже кода можно выбрать из датафрейма столбцы `name`, `color`, `origin`, `weight` (в килограммах) и `is_round`:

```
fruit.select(
    pl.col("name"), ❶
    pl.col("^.*or.*$"), ❷
    pl.col("weight") / 1000, ❸
    "is_round", ❹
)
```

❶ Применение функции *`pl.col()`* является наиболее распространенным подходом к созданию выражений[1]. В качестве аргумента мы можем передать строковое представление существующего столбца – в данном случае `"name"`.

❷ Функция `pl.col()` также может принимать в качестве аргумента регулярное выражение, о чем мы упоминали выше. Использованное в этом примере регулярное выражение (`"^.*or.*$"`) подходит сразу к двум столбцам (`color` и `origin`), поскольку они оба содержат подстроку `or`.

❸ Вы можете применять арифметические операции (сложение, вычитание и т. д.) к выражениям с использованием уже знакомых вам операторов (подробнее об арифметических вычислениях применительно к выражениям мы будем говорить

[1] Чисто технически `pl.col()` не является функцией, но мы будем ее так называть, поскольку использоваться она может как функция.

в главе 9). Обратите внимание, что Polars автоматически привел целочисленный тип столбца `weight` (`i64`) к вещественному (`f64`) для выполнения операции деления.

❹ Метод `select()` также может принимать строковое представление существующего столбца. Этот способ может быть удобен своей лаконичностью. В то же время, поскольку строка не является выражением, в этом случае вы не сможете применить к столбцу арифметические и прочие операции.

Вывод:

```
shape: (10, 5)
┌─────────────┬────────┬─────────────────┬────────┬──────────┐
│ name        ┆ color  ┆ origin          ┆ weight ┆ is_round │
│ ---         ┆ ---    ┆ ---             ┆ ---    ┆ ---      │
│ str         ┆ str    ┆ str             ┆ f64    ┆ bool     │
╞═════════════╪════════╪═════════════════╪════════╪══════════╡
│ Avocado     ┆ green  ┆ South America   ┆ 0.2    ┆ false    │
│ Banana      ┆ yellow ┆ Asia            ┆ 0.12   ┆ false    │
│ Blueberry   ┆ blue   ┆ North America   ┆ 0.001  ┆ false    │
│ Cantaloupe  ┆ orange ┆ Africa          ┆ 2.5    ┆ true     │
│ Cranberry   ┆ red    ┆ North America   ┆ 0.002  ┆ false    │
│ Elderberry  ┆ black  ┆ Europe          ┆ 0.001  ┆ false    │
│ Orange      ┆ orange ┆ Asia            ┆ 0.13   ┆ true     │
│ Papaya      ┆ orange ┆ South America   ┆ 1.0    ┆ false    │
│ Peach       ┆ orange ┆ Asia            ┆ 0.15   ┆ true     │
│ Watermelon  ┆ green  ┆ Africa          ┆ 5.0    ┆ true     │
└─────────────┴────────┴─────────────────┴────────┴──────────┘
```

## Создание новых столбцов с помощью выражений

Метод датафрейма *`with_columns()`* позволяет создать один или несколько столбцов как на основе существующих столбцов, так и с нуля, и добавить их к датафрейму. В примере, приведенном ниже, мы добавим два новых столбца в наш датафрейм `fruit`: первый будет служить индикатором того, что плод является фруктом (и для всех наших строк установим значение `True`), второй – что плод принадлежит к ягодам (к счастью, в английском языке названия ягод в основном оканчиваются на *berry*):

```
fruit.with_columns(
    pl.lit(True).alias("is_fruit"), ❶
    is_berry=pl.col("name").str.ends_with("berry"), ❷
)
```

❶ С помощью функции *`pl.lit()`* можно создать выражение на основе литерального значения, такого как `True`. Метод *`Expr.alias()`* позволяет задавать имена новых столбцов и переименовывать существующие.

❷ Метод *`Expr.str.ends_with()`* является одним из множества методов, принадлежащих пространству имен `str`. Подробнее об этом мы будем говорить в главе 12. В данном случае мы воспользовались синтаксисом на основе ключевого аргумента (`is_berry=`) вместо вызова метода `Expr.alias()` для указания имени нового столбца (`is_berry`).

Вывод:

```
shape: (10, 7)
```

| name | weight | color | is_round | origin | is_fruit | is_berry |
|---|---|---|---|---|---|---|
| str | i64 | str | bool | str | bool | bool |
| Avocado | 200 | green | false | South Amer… | true | false |
| Banana | 120 | yellow | false | Asia | true | false |
| Blueberry | 1 | blue | false | North Amer… | true | true |
| Cantaloupe | 2500 | orange | true | Africa | true | false |
| Cranberry | 2 | red | false | North Amer… | true | true |
| Elderberry | 1 | black | false | Europe | true | true |
| Orange | 130 | orange | true | Asia | true | false |
| Papaya | 1000 | orange | false | South Amer… | true | false |
| Peach | 150 | orange | true | Asia | true | false |
| Watermelon | 5000 | green | true | Africa | true | false |

### Именование столбцов

В предыдущих примерах мы использовали два способа указания имен для столбцов.

Первый способ состоит в применении метода `Expr.alias()` для переименования выражения, что ведет к определению имени для столбца. В данном случае имя привязано к самому выражению вне зависимости от того, где оно используется.

Второй способ состоит в использовании синтаксиса на основе ключевого аргумента для установки имени для нового столбца напрямую. В этом случае имя является локальным в контексте, в котором используется выражение.

В некоторых ситуациях применение синтаксиса на основе ключевого аргумента бывает невозможно. После использования ключевого аргумента все остальные аргументы функции (выражения) также должны быть ключевыми. А мы знаем, что ключевые аргументы в Python должны отвечать определенным требованиям именования: не могут начинаться с цифры, должны содержать только буквенно-числовые символы и знаки подчеркивания и не могут совпадать с зарезервированными словами в Python, такими как `class` или `except`.

При использовании обоих способов вариант с ключевыми аргументами будет обладать приоритетом над именованием выражений. Мы в большинстве случаев предпочитаем применять именно ключевые аргументы по причине ясности и лаконичности такого подхода.

## Фильтрация строк при помощи выражений

Для осуществления фильтрации строк на основе выражений можно воспользоваться методом датафрейма *`filter()`*. В результирующем наборе будут сохранены только те строки, для которых вычисление выражения дает `True`. На-

пример, оставить в датафрейме только круглые фрукты весом более одного килограмма можно так:

```
fruit.filter(
    (pl.col("weight") > 1000) ❶
    & pl.col("is_round") ❷
)
```

❶ Существующие столбцы в датафрейме можно преобразовать в булевы при помощи операторов сравнения, таких как < (меньше) или > (больше).
❷ Здесь мы скомбинировали два выражения посредством логического оператора И (`&`). В результате мы получим значение `True` только в случае, если оба выражения вернут `True` (подробнее о логических операторах мы будем говорить в главе 9).

Вывод:

```
shape: (2, 5)
┌─────────────┬────────┬────────┬──────────┬────────┐
│ name        ┆ weight ┆ color  ┆ is_round ┆ origin │
│ ---         ┆ ---    ┆ ---    ┆ ---      ┆ ---    │
│ str         ┆ i64    ┆ str    ┆ bool     ┆ str    │
╞═════════════╪════════╪════════╪══════════╪════════╡
│ Cantaloupe  ┆ 2500   ┆ orange ┆ true     ┆ Africa │
│ Watermelon  ┆ 5000   ┆ green  ┆ true     ┆ Africa │
└─────────────┴────────┴────────┴──────────┴────────┘
```

## Агрегирование данных при помощи выражений

Операции агрегирования данных в основном сводятся к созданию групп строк с дальнейшим их укрупнением до одной строки по неким заданным правилам. В примере ниже мы создаем группы с помощью метода датафрейма *`group_by()`* на основе последнего слова в столбце `origin`, после чего рассчитываем количество фруктов, вошедших в каждую группу, и их средний вес. Обратите внимание, что здесь выражения используются в двух разных местах кода: при определении групп и при их агрегации:

```
fruit.group_by(pl.col("origin").str.split(" ").list.last()).agg( ❶
    pl.len(), ❷
    average_weight=pl.col("weight").mean() ❸
)
```

❶ Каждое уникальное значение этого выражения (на основе последнего слова в столбце `origin`) выделяется в обособленную группу.
❷ Выражение, созданное при помощи функции *`pl.len()`*, возвращает количество строк в каждой группе.
❸ Метод *`Expr.mean()`* применяется для укрупнения нескольких значений путем вычисления среднего арифметического.

Вывод:

```
shape: (4, 3)
┌──────────┬─────┬──────────────────┐
│ origin   ┆ len ┆ average_weight   │
│ ---      ┆ --- ┆ ---              │
│ str      ┆ u32 ┆ f64              │
╞══════════╪═════╪══════════════════╡
│ America  ┆ 4   ┆ 300.75           │
│ Asia     ┆ 3   ┆ 133.333333       │
│ Africa   ┆ 2   ┆ 3750.0           │
│ Europe   ┆ 1   ┆ 1.0              │
└──────────┴─────┴──────────────────┘
```

**Впечатляющая параллельность**

Мы не хотели бы забегать далеко вперед, но нас распирает от желания поскорее сообщить вам о том, что несколько выражений в Polars могут выполняться параллельно, как в наших примерах с отбором столбцов и агрегированием значений. Это одна из причин такого ошеломляющего быстродействия библиотеки Polars.

# Сортировка строк при помощи выражений

Для изменения порядка следования строк в датафрейме на основе одного или нескольких столбцов можно воспользоваться методом *`sort()`*. В приведенном ниже примере мы отсортируем строки в таблице по длине наименования фруктов:

```
fruit.sort(
    pl.col("name").str.len_bytes(), ❶ ❷
    descending=True, ❸
)
```

❶ В пространстве имен `str` отсутствует метод `len()`, поскольку определять длину содержимого строк можно двумя способами: в байтах (*`len_bytes()`*) или в символах (*`len_chars()`*).

Пример из официальной документации, демонстрирующий разницу между методами `len_bytes()` и `len_chars()`:

```
df = pl.DataFrame({"a": ["Café", "345", "東京", None]})
df.with_columns(
    pl.col("a").str.len_bytes().alias("n_bytes"),
    pl.col("a").str.len_chars().alias("n_chars"),
)
```

Вывод:

```
shape: (4, 3)
┌──────┬─────────┬─────────┐
│ a    ┆ n_bytes ┆ n_chars │
│ ---  ┆ ---     ┆ ---     │
│ str  ┆ u32     ┆ u32     │
╞══════╪═════════╪═════════╡
```

```
| Café | 5      | 4      |
| 345  | 3      | 3      |
| 東京 | 6      | 2      |
| null | null   | null   |
└──────┴────────┴────────┘
```

❷ Вы можете сортировать строки на основе выражения, не присутствующего в датафрейме. В нашем случае имена фруктов в датафрейме присутствуют, а длины имен – нет. Нет никакой необходимости явно добавлять столбец в датафрейм, если вам нужно просто упорядочить данные по нему.

❸ Для сортировки в порядке возрастания (по умолчанию) достаточно просто удалить аргумент `descending` или передать ему значение `False`.

Вывод:

```
shape: (10, 5)
```

| name | weight | color | is_round | origin |
|---|---|---|---|---|
| str | i64 | str | bool | str |
| Cantaloupe | 2500 | orange | true | Africa |
| Elderberry | 1 | black | false | Europe |
| Watermelon | 5000 | green | true | Africa |
| Blueberry | 1 | blue | false | North America |
| Cranberry | 2 | red | false | North America |
| Avocado | 200 | green | false | South America |
| Banana | 120 | yellow | false | Asia |
| Orange | 130 | orange | true | Asia |
| Papaya | 1000 | orange | false | South America |
| Peach | 150 | orange | true | Asia |

# Определение выражения

Теперь, когда мы рассмотрели несколько примеров использования выражений в Polars, пришло время познакомиться с формальным, но очень важным определением того, что на самом деле представляют из себя выражения.

> Выражение – это дерево операций, описывающее правила построения одного или более объектов Series.

Это определение весьма емкое, но при этом каждое слово в нем имеет огромное значение. Давайте разберем его на пять ключевых частей.

## *Объекты Series*

Как мы помним из главы 4, объект Series представляет собой массив значений с одним типом данных, таким как `pl.Float64` для 64-битных веществен-

ных чисел или `pl.String` – для строк. Вы можете думать об объектах Series как о столбцах в датафрейме, но не стоит забывать, что они могут существовать и сами по себе и далеко *не всегда* являются частью датафрейма.

### *Дерево операций*

Выражения могут состоять как из одной операции, так и из нескольких, обладающих либо линейной последовательностью, либо сложной древовидной структурой.

На рис. 7.2 показаны три примера выражений. При вычислении этих выражений они выдадут три столбца со значениями 3, 8 и 4 соответственно. Обратите внимание, что третья диаграмма на рисунке обладает древовидной структурой.

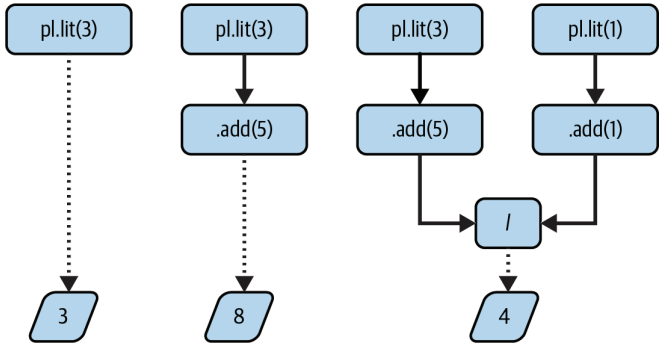



**Рис. 7.2** ❖ Выражение – это дерево операций

Если говорить в общем, все выражения обладают древовидной структурой, но при этом они не обязаны иметь ветви и родительские элементы.

### *Описывающее правила*

Выражение – это не более чем описание. Само по себе выражение не способно ни создать объект Series, ни выполниться самостоятельно за неимением соответствующих методов. Выражения вычисляются только в результате передачи их в виде аргументов в функции вроде `pl.select()` или методов, таких как `group_by()`. И только затем мы получаем один или несколько объектов Series.

Если представить выражение в виде рецепта, то операции будут шагами в этом рецепте, а функции и методы, в которые передается выражение, – поварами.

### *Построение*

Вы не всегда будете видеть построенный в результате применения выражения объект Series. Будет ли он добавляться в датафрейм в виде отдельного столбца, зависит от конкретной функции или метода, вычисляющего выражение. Примером того, когда объект Series не включается в результирующий

датафрейм в виде столбца, является функция *pl.filter()*. При ее использовании этот объект применяется только для определения того, какие строки сохранятся в результирующем датафрейме. А новые колонки не добавляются. Но, к слову, к *построению* также стоит относиться с осторожностью: если выражение состоит из единственной операции, которая ссылается на существующий столбец, никакой объект Series даже не создается.

### *Одного или более*

С помощью одного выражения можно построить более одного объекта Series. К примеру, функция *pl.all()* ссылается на все столбцы в датафрейме сразу. Таким образом, в результате вычисления выражения `pl.all().mul(10).name.suffix("_times_10")` значения во всех столбцах датафрейма будут умножены на 10, а к именам столбцов будет добавлена строка `_times_10`:

```
(
    pl.DataFrame({"a": [1, 2, 3], "b": [0.4, 0.5, 0.6]}).with_columns(
        pl.all().mul(10).name.suffix("_times_10")
    )
)
```

Вывод:

```
shape: (3, 4)
┌─────┬─────┬────────────┬────────────┐
│ a   ┆ b   ┆ a_times_10 ┆ b_times_10 │
│ --- ┆ --- ┆ ---        ┆ ---        │
│ i64 ┆ f64 ┆ i64        ┆ f64        │
╞═════╪═════╪════════════╪════════════╡
│ 1   ┆ 0.4 ┆ 10         ┆ 4.0        │
│ 2   ┆ 0.5 ┆ 20         ┆ 5.0        │
│ 3   ┆ 0.6 ┆ 30         ┆ 6.0        │
└─────┴─────┴────────────┴────────────┘
```

С помощью метода *Expr.meta.has_multiple_outputs()* можно проверить, может ли в результате вычисления выражения образоваться более одного объекта Series. Пространство имен `Expr.meta` мы будем подробно обсуждать в главе 18:

```
pl.all().mul(10).name.suffix("_times_10").meta.has_multiple_outputs()
```

Вывод:

```
True
```

Возможность построения более одного объекта Series зависит от датафрейма, к которому применяется выражение. Если в датафрейме содержится только один столбец, то и вычисление выражения `pl.all()` приведет к образованию единственного объекта Series.

# Свойства выражений

Одно дело – знать определение выражений, и совсем другое – понимать, как они в действительности работают. Ниже мы приведем несколько важных свойств выражений.

## *Ленивое вычисление*

Выражения – ленивые создания по своей натуре. Пока их хорошо не попросишь, сами они ничего не сделают. Возможно, это наиболее важная их характеристика, поскольку без нее не были бы реализуемы следующие четыре свойства, о которых речь пойдет далее.

## *Зависимость от функции и данных*

Выражения одновременно зависят как от функций, которые их выполняют, так и от датафрейма (обычного или ленивого), к которому применяются. Функция определяет, что будет происходить с созданным объектом Series, а датафрейм – тип и длину объекта.

Для демонстрации сказанного давайте передадим одно и то же выражение `is_orange` трем различным функциям (методам) и посмотрим на вывод:

```
is_orange = (pl.col("color") == "orange").alias("is_orange")

fruit.with_columns(is_orange)
```

Вывод:

```
shape: (10, 6)
┌────────────┬────────┬────────┬──────────┬───────────────┬───────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin        ┆ is_orange │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---           ┆ ---       │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str           ┆ bool      │
╞════════════╪════════╪════════╪══════════╪═══════════════╪═══════════╡
│ Avocado    ┆ 200    ┆ green  ┆ false    ┆ South America ┆ false     │
│ Banana     ┆ 120    ┆ yellow ┆ false    ┆ Asia          ┆ false     │
│ Blueberry  ┆ 1      ┆ blue   ┆ false    ┆ North America ┆ false     │
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa        ┆ true      │
│ Cranberry  ┆ 2      ┆ red    ┆ false    ┆ North America ┆ false     │
│ Elderberry ┆ 1      ┆ black  ┆ false    ┆ Europe        ┆ false     │
│ Orange     ┆ 130    ┆ orange ┆ true     ┆ Asia          ┆ true      │
│ Papaya     ┆ 1000   ┆ orange ┆ false    ┆ South America ┆ true      │
│ Peach      ┆ 150    ┆ orange ┆ true     ┆ Asia          ┆ true      │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa        ┆ false     │
└────────────┴────────┴────────┴──────────┴───────────────┴───────────┘
```

```
fruit.filter(is_orange)
```

Вывод:

```
shape: (4, 5)
┌─────────────┬────────┬────────┬──────────┬───────────────┐
│ name        ┆ weight ┆ color  ┆ is_round ┆ origin        │
│ ---         ┆ ---    ┆ ---    ┆ ---      ┆ ---           │
│ str         ┆ i64    ┆ str    ┆ bool     ┆ str           │
╞═════════════╪════════╪════════╪══════════╪═══════════════╡
│ Cantaloupe  ┆ 2500   ┆ orange ┆ true     ┆ Africa        │
│ Orange      ┆ 130    ┆ orange ┆ true     ┆ Asia          │
│ Papaya      ┆ 1000   ┆ orange ┆ false    ┆ South America │
│ Peach       ┆ 150    ┆ orange ┆ true     ┆ Asia          │
└─────────────┴────────┴────────┴──────────┴───────────────┘
```

```
fruit.group_by(is_orange).len()
```

Вывод:

```
shape: (2, 2)
┌────────────┬─────┐
│ is_orange  ┆ len │
│ ---        ┆ --- │
│ bool       ┆ u32 │
╞════════════╪═════╡
│ false      ┆ 6   │
│ true       ┆ 4   │
└────────────┴─────┘
```

Ключевая идея состоит в том, что вы можете использовать один и тот же синтаксис для выполнения разных задач. Это приводит нас к следующему свойству выражений.

### *Повторная используемость*

Выражения представляют собой выражения Python. В предыдущем примере мы создали объект выражения в переменной `is_orange` и повторно использовали его, передавая разным методам датафрейма `fruit`. Дальше – больше, поскольку никто не запрещает нам использовать созданное выражение и применительно к другим датафреймам:

```
flowers = pl.DataFrame(
    {
        "name": ["Tiger lily", "Blue flag", "African marigold"],
        "latin": ["Lilium columbianum", "Iris versicolor", "Tagetes erecta"],
        "color": ["orange", "purple", "orange"],
    }
)
```

```
flowers.filter(is_orange)
```

Вывод:

```
shape: (2, 3)
┌──────────────────┬──────────────────────┬────────┐
│ name             ┆ latin                ┆ color  │
│ ---              ┆ ---                  ┆ ---    │
│ str              ┆ str                  ┆ str    │
╞══════════════════╪══════════════════════╪════════╡
│ Tiger lily       ┆ Lilium columbianum   ┆ orange │
│ African marigold ┆ Tagetes erecta       ┆ orange │
└──────────────────┴──────────────────────┴────────┘
```

### *Выразительность*

Как бы странно это ни казалось, выражения в Polars выразительны, и на то есть две причины. Во-первых, одно выражение может оперировать с несколькими объектами Series (столбцами) одновременно, что позволяет создавать мощные и гибкие преобразования без необходимости повторять уже написанный ранее код. Во-вторых, выражения обладают поразительной гибкостью, что дает возможность объединять и комбинировать в цепочки сложные операции, связанные с арифметическими вычислениями и манипуляциями со строками. Такая выразительность позволяет очень ясно и понятно описывать производимые с помощью выражений действия.

### *Эффективность*

Из-за следования ленивой парадигме выражения поддаются оптимизации перед их выполнением. Polars минимизирует количество вычислений, необходимых для создания требуемого объекта Series путем анализа операций, входящих в выражение. Более того, и это очень важно, когда функции передается несколько выражений, она может вычислять их параллельно.

Как видите, выражения обладают рядом очень важных свойств. Давайте, наконец, посмотрим, как можно создавать эти магические выражения.

## Создание выражений

Каждое выражение начинается с операции. Вообще говоря, новое выражение создается с помощью функции, не зависящей от других выражений. Получив выражение, вы можете наращивать его посредством других методов и комбинировать с другими выражениями при помощи встраиваемых операторов (о них мы будем подробно говорить в следующих двух главах). Давайте рассмотрим способы создания выражений в Polars.

## На основе существующих столбцов

Наиболее распространенный способ создания выражений базируется на обращении к одному или нескольким существующим столбцам в датафрейме. И это понятно, ведь в большинстве случаев нам приходится преобразовывать имеющиеся у нас данные. Это можно сделать при помощи функции *pl.col()*, которая может принимать имена столбцов, регулярные выражения или типы данных. Рассмотрим несколько примеров.

Для демонстрации мы воспользуемся методом датафрейма `select()` для выбора столбцов и атрибутом `columns` для получения списка имен колонок. К конкретному столбцу вы можете обратиться по имени следующим образом:

```
fruit.select(pl.col("color")).columns
```

Вывод:

```
['color']
```

Если в датафрейме отсутствует колонка с таким именем, Polars выдаст исключение:

```
fruit.select(pl.col("is_smelly")).columns
```

Вывод:

```
ColumnNotFoundError: is_smelly

Resolved plan until failure:

        ---> FAILED HERE RESOLVING 'select' <---
DF ["name", "weight", "color", "is_round"]; PROJECT */5 COLUMNS
```

*Регулярные выражения* (regular expression) могут быть весьма полезны при обращении сразу к нескольким столбцам, имена которых следуют определенному шаблону. При этом вам необходимо начать регулярное выражение с символа крышки (^) и завершить символом доллара ($), как показано ниже:

```
fruit.select(pl.col("^.*or.*$")).columns
```

Вывод:

```
['color', 'origin']
```

С помощью функции `pl.col("*")` или ее удобного алиаса `pl.all()` вы можете сослаться сразу на все столбцы в датафрейме:

```
fruit.select(pl.all()).columns
```

Вывод:

```
['name', 'weight', 'color', 'is_round', 'origin']
```

Также вы можете обратиться ко всем столбцам определенного типа, переданного функции `pl.col()` в виде аргумента. Например, так вы можете выбрать в датафрейме все столбцы со строковым содержимым:

```
fruit.select(pl.col(pl.String)).columns
```

Вывод:

```
['name', 'color', 'origin']
```

Кроме того, вы можете передать функции `pl.col()` сразу несколько имен столбцов или типов данных:

```
fruit.select(pl.col(pl.Boolean, pl.Int64)).columns
```

Вывод:

```
['weight', 'is_round']
```

Также их можно передать списком, что бывает весьма удобно:

```
fruit.select(pl.col(["name", "color"])).columns
```

Вывод:

```
['name', 'color']
```

**Селекторы столбцов**

Вы можете создавать выражения в Polars при помощи так называемых селекторов столбцов. Этот подход является рекомендованным при необходимости выбора нескольких столбцов на основе их свойств (например, для выбора всех числовых столбцов или столбцов определенного типа). Подробно об этом мы поговорим в главе 10.

## На основе литеральных значений

Для создания нового выражения на основе некоторого значения в Python можно воспользоваться функцией *`pl.lit()`*. *Lit* является сокращением от *literal*. В следующих примерах мы продемонстрируем создание датафреймов с нуля при помощи функции pl.select() (заметьте, что это именно функция, а не метод датафрейма с таким же названием, который мы использовали ранее):

```
pl.select(pl.lit(42))
```

Вывод:

```
shape: (1, 1)
┌─────────┐
│ literal │
│ ---     │
```

```
| i32     |
╞═════════╡
| 42      |
└─────────┘
```

Обратите внимание на имя столбца `literal`. Вы можете дать столбцу и более благозвучное имя при помощи метода *`Expr.alias()`*:

```
pl.select(pl.lit(42).alias("answer"))
```

Вывод:

```
shape: (1, 1)
┌────────┐
| answer |
| ---    |
| i32    |
╞════════╡
| 42     |
└────────┘
```

Также вы можете задать имя столбца при помощи ключевого аргумента, как показано ниже:

```
pl.select(answer=pl.lit(42))
```

При вычислении показанных выше выражений посредством функции `pl.select()` мы получаем на выходе объект Series с единственным значением. Но если применить ту же операцию к существующему датафрейму, длина итогового объекта Series будет соответствовать количеству строк в датафрейме, поскольку все колонки в датафрейме должны быть одной длины:

```
fruit.with_columns(planet=pl.lit("Earth"))
```

Вывод:

```
shape: (10, 6)
┌─────────────┬────────┬────────┬──────────┬─────────────────┬────────┐
| name        | weight | color  | is_round | origin          | planet |
| ---         | ---    | ---    | ---      | ---             | ---    |
| str         | i64    | str    | bool     | str             | str    |
╞═════════════╪════════╪════════╪══════════╪═════════════════╪════════╡
| Avocado     | 200    | green  | false    | South America   | Earth  |
| Banana      | 120    | yellow | false    | Asia            | Earth  |
| Blueberry   | 1      | blue   | false    | North America   | Earth  |
| Cantaloupe  | 2500   | orange | true     | Africa          | Earth  |
| Cranberry   | 2      | red    | false    | North America   | Earth  |
| Elderberry  | 1      | black  | false    | Europe          | Earth  |
| Orange      | 130    | orange | true     | Asia            | Earth  |
| Papaya      | 1000   | orange | false    | South America   | Earth  |
| Peach       | 150    | orange | true     | Asia            | Earth  |
| Watermelon  | 5000   | green  | true     | Africa          | Earth  |
└─────────────┴────────┴────────┴──────────┴─────────────────┴────────┘
```

Как видите, значение Earth было продублировано во всех строках датафрейма. Это возможно при передаче функции pl.lit() единственного значения. Но если вы передадите ей больше одного значения, но меньше, чем количество строк в датафрейме, возникнет ошибка:

```
fruit.with_columns(pl.lit(pl.Series([False, True])).alias("row_is_even"))
```

Вывод:

```
ShapeError: unable to add a column of length 2 to a DataFrame of height 10
```

Как видите, сначала мы преобразовали список [False, True] в объект Series с помощью конструктора *pl.Series()*. Если бы мы этого не сделали, был бы создан столбец со списками в виде значений, как показано ниже:

```
fruit.with_columns(row_is_even=pl.lit([False, True]))
```

Вывод:

```
shape: (10, 6)
```

| name | weight | color | is_round | origin | row_is_even |
|---|---|---|---|---|---|
| str | i64 | str | bool | str | list[bool] |
| Avocado | 200 | green | false | South America | [false, true] |
| Banana | 120 | yellow | false | Asia | [false, true] |
| … | … | … | … | … | … |
| Peach | 150 | orange | true | Asia | [false, true] |
| Watermelon | 5000 | green | true | Africa | [false, true] |

Для повторения значений фиксированное количество раз вы можете воспользоваться удобной функцией *pl.repeat()*. Функции *pl.zeros()* и *pl.ones()* являются сокращенными способами вызова функций pl.repeat(0.0) и pl.repeat(1.0) соответственно:

```
pl.select(pl.repeat("Ella", 3).alias("umbrella"), pl.zeros(3), pl.ones(3))
```

Вывод:

```
shape: (3, 3)
```

| umbrella | zeros | ones |
|---|---|---|
| str | f64 | f64 |
| Ella | 0.0 | 1.0 |
| Ella | 0.0 | 1.0 |
| Ella | 0.0 | 1.0 |

Обратите внимание, что длины всех объектов Series должны совпадать, иначе возникнет ошибка:

```
fruit.with_columns(planet=pl.repeat("Earth", 9))
```

Вывод:

```
ShapeError: unable to add a column of length 9 to a DataFrame of height 10
```

## На основе диапазонов

В Polars присутствует несколько удобных функций для создания диапазонов числовых и календарных значений. Они перечислены в табл. 7.1.

***Таблица 7.1. Функции для создания диапазонов значений***

| Функция | Описание |
|---|---|
| *pl.arange()* | Диапазон целочисленных значений. Алиас функции `pl.int_range()` |
| *pl.date_range()* | Диапазон дат |
| *pl.date_ranges()* | Каждый элемент – диапазон дат |
| *pl.datetime_range()* | Диапазон дат и времен |
| *pl.datetime_ranges()* | Каждый элемент – диапазон дат и времен |
| *pl.int_range()* | Диапазон целочисленных значений |
| *pl.int_ranges()* | Каждый элемент – диапазон целочисленных значений |
| *pl.time_range()* | Диапазон времен |
| *pl.time_ranges()* | Каждый элемент – диапазон времен |

В следующем примере мы продемонстрируем работу функции `pl.int_range()`, ее алиаса `pl.arange()` и `pl.int_ranges()`. Также мы воспользуемся методом *`Expr.list.len()`*, с помощью которого посчитаем количество элементов в каждом списке в столбце `int_range`:

```
pl.select(
    start=pl.int_range(0, 5), end=pl.arange(0, 10, 2).pow(2)
).with_columns(int_range=pl.int_ranges("start", "end")).with_columns(
    range_length=pl.col("int_range").list.len()
)
```

Вывод:

```
shape: (5, 4)
┌───────┬─────┬─────────────┬──────────────┐
│ start ┆ end ┆ int_range   ┆ range_length │
│ ---   ┆ --- ┆ ---         ┆ ---          │
│ i64   ┆ i64 ┆ list[i64]   ┆ u32          │
╞═══════╪═════╪═════════════╪══════════════╡
│ 0     ┆ 0   ┆ []          ┆ 0            │
```

```
| 1     | 4   | [1, 2, 3]    | 3           |
| 2     | 16  | [2, 3, … 15] | 14          |
| 3     | 36  | [3, 4, … 35] | 33          |
| 4     | 64  | [4, 5, … 63] | 60          |
└───────┴─────┴──────────────┴─────────────┘
```

Заметьте, что функция `pl.int_ranges()` генерирует объект Series, каждым элементом которого является список целочисленных значений. Функции *`pl.date_ranges()`*, *`pl.datetime_ranges()`* и *`pl.time_ranges()`* работают аналогично, только со значениями календарных типов:

```
pl.select(
    start=pl.date_range(pl.date(1985, 10, 21), pl.date(1985, 10, 26)),
    end=pl.repeat(pl.date(2021, 10, 21), 6),
).with_columns(range=pl.datetime_ranges("start", "end", interval="1h"))
```

Вывод:

```
shape: (6, 3)
┌────────────┬────────────┬──────────────────────────────────────────────┐
| start      | end        | range                                        |
| ---        | ---        | ---                                          |
| date       | date       | list[datetime[μs]]                           |
╞════════════╪════════════╪══════════════════════════════════════════════╡
| 1985-10-21 | 2021-10-21 | [1985-10-21 00:00:00, 1985-10-21 01:00:00, … |
| 1985-10-22 | 2021-10-21 | [1985-10-22 00:00:00, 1985-10-22 01:00:00, … |
| 1985-10-23 | 2021-10-21 | [1985-10-23 00:00:00, 1985-10-23 01:00:00, … |
| 1985-10-24 | 2021-10-21 | [1985-10-24 00:00:00, 1985-10-24 01:00:00, … |
| 1985-10-25 | 2021-10-21 | [1985-10-25 00:00:00, 1985-10-25 01:00:00, … |
| 1985-10-26 | 2021-10-21 | [1985-10-26 00:00:00, 1985-10-26 01:00:00, … |
└────────────┴────────────┴──────────────────────────────────────────────┘
```

В главе 12 мы более подробно коснемся темы работы с календарными типами данных в Polars.

## Другие функции для создания измерений

Существует немало функций для создания выражений. К сожалению, мы не сможем рассказать о них всех в этой главе. Но мы покажем вам несколько из них с указанием того, где можно почитать о них подробнее.

Функция *`pl.len()`*, как ясно из названия, используется для подсчета количества элементов в столбце. Ее часто применяют при использовании агрегации методом `group_by()`. Подробнее об этом можно почитать в главе 13.

Также есть любопытная функция *`pl.element()`*, служащая для представления одного элемента списка. Она используется совместно с методом *`Expr.list.eval()`* для применения выражения к каждому элементу в списке. В детали мы погрузимся в главе 12.

# Переименование выражений

Операция переименования выражений, которая в конечном счете определяет имя создаваемого объекта Series, применяется в Polars довольно часто. Существует масса причин для переименования выражений, включая следующие:

- чтобы облегчить пользователю понимание о том, что содержится в столбце;
- чтобы избежать дублирования имен столбцов;
- чтобы очистить имена столбцов;
- чтобы изменить имя столбца по умолчанию.

**Подходящие имена**

Подбирать для выражений подходящие имена не менее важно, чем для переменных. Этот аспект может существенно сказаться на легкости восприятия вашего кода. Мы рекомендуем использовать в именах столбцов исключительно строчные буквы с символами подчеркивания между словами. Такой стиль наименования называется змеиным.

Наиболее часто употребимым методом для переименования выражений является *`Expr.alias()`*. Другие полезные методы для работы с именами выражений собраны в пространстве имен *`Expr.name`* и показаны в табл. 7.2. Методы `Expr.name.map_fields()`, `Expr.name.prefix_fields()` и `Expr.name.suffix_fields()` могут использоваться только с выражениями с типом данных `pl.Struct`.

***Таблица 7.2. Методы для переименования выражений***

| Функция | Описание |
|---|---|
| *`Expr.alias()`* | Переименовывает выражение |
| *`Expr.name.keep()`* | Сохраняет исходное имя выражения, отменяя при этом все ранее выполненные операции по переименованию |
| *`Expr.name.map()`* | Переименовывает выражение путем применения функции к исходному имени |
| *`Expr.name.prefix()`* | Добавляет префикс к исходному имени столбца выражения |
| *`Expr.name.suffix()`* | Добавляет суффикс к исходному имени столбца выражения |
| *`Expr.name.to_lowercase()`* | Приводит исходное имя выражения к нижнему регистру |
| *`Expr.name.to_uppercase()`* | Приводит исходное имя выражения к верхнему регистру |
| *`Expr.name.map_fields()`* | Переименовывает поля структуры путем применения функции к именам полей |
| *`Expr.name.prefix_fields()`* | Добавляет префикс ко всем именам полей структуры |
| *`Expr.name.suffix_fields()`* | Добавляет суффикс ко всем именам полей структуры |

Для иллюстрации рассмотрим пример с небольшим датафреймом с произвольными именами столбцов:

```
df = pl.DataFrame({"text": "value", "An integer": 5040, "BOOLEAN": True})
df
```

Вывод:

```
shape: (1, 3)
┌──────┬────────────┬─────────┐
│ text ┆ An integer ┆ BOOLEAN │
│ ---  ┆ ---        ┆ ---     │
│ str  ┆ i64        ┆ bool    │
╞══════╪════════════╪═════════╡
│ value ┆ 5040      ┆ true    │
└──────┴────────────┴─────────┘
```

Мы можем изменить имена столбцов в этом датафрейме разными способами:

```
df.select(
    pl.col("text").name.to_uppercase(),
    pl.col("An integer").alias("int"),
    pl.col("BOOLEAN").name.to_lowercase(),
)
```

Вывод:

```
shape: (1, 3)
┌──────┬─────┬─────────┐
│ TEXT ┆ int ┆ boolean │
│ ---  ┆ ---  ┆ ---     │
│ str  ┆ i64  ┆ bool    │
╞══════╪═════╪═════════╡
│ value ┆ 5040 ┆ true    │
└──────┴─────┴─────────┘
```

**Объединение операций именования в цепочки**

На момент написания книги в Polars было допустимо использовать только одну операцию именования для каждого выражения. Таким образом, следующий код выдаст ошибку:

```
df.select(
    pl.all()
    .name.to_lowercase()
    .name.map(lambda s: s.replace(" ", "_"))
)
```

Вывод:

```
InvalidOperationError: 'Expr: .rename_alias(col("text"))'
not allowed in this context/location

Resolved plan until failure:

            ---> FAILED HERE RESOLVING 'select' <---
DF ["text", "An integer", "BOOLEAN"]; PROJECT */3 COLUMNS
```

Решение можно найти в объединении всех нужных операций в одну анонимную функцию и ее применении к выражению с помощью метода `Expr.name.map()`, как показано ниже:

```
df.select(
    pl.all().name.map(lambda s: s.lower().replace(" ", "_"))
)
```

Вывод:

```
shape: (1, 3)
┌──────┬────────────┬─────────┐
│ text ┆ an_integer ┆ boolean │
│ ---  ┆ ---        ┆ ---     │
│ str  ┆ i64        ┆ bool    │
╞══════╪════════════╪═════════╡
│ value ┆ 5040      ┆ true    │
└──────┴────────────┴─────────┘
```

В следующих версиях Polars это ограничение может быть снято.

# Выражения – характерная черта Polars

Вы уже знаете, что выражения ленивы по своей натуре и требуют внешнего воздействия для выполнения. Мы понимаем, что привыкнуть к выражениям может быть непросто, особенно если у вас есть достаточно большой опыт работы с фреймворками, следующими жадной парадигме вычислений, такими как pandas.

И здесь есть одна опасность. Все действия, осуществляемые при помощи методов выражений и встраиваемых операторов, можно также реализовать посредством работы с объектами Series. К примеру, код для фильтрации датафрейма, который мы написали ранее с использованием выражений, может быть переписан следующим образом:

```
fruit.filter((fruit["weight"] > 1000) & fruit["is_round"])
```

Вывод:

```
shape: (2, 5)
┌─────────────┬────────┬────────┬──────────┬────────┐
│ name        ┆ weight ┆ color  ┆ is_round ┆ origin │
│ ---         ┆ ---    ┆ ---    ┆ ---      ┆ ---    │
│ str         ┆ i64    ┆ str    ┆ bool     ┆ str    │
╞═════════════╪════════╪════════╪══════════╪════════╡
│ Cantaloupe  ┆ 2500   ┆ orange ┆ true     ┆ Africa │
│ Watermelon  ┆ 5000   ┆ green  ┆ true     ┆ Africa │
└─────────────┴────────┴────────┴──────────┴────────┘
```

Напомним, что код с использованием выражений выглядел так:

```
fruit.filter(
    (pl.col("weight") > 1000)
    & pl.col("is_round")
)
```

Если вы работали с библиотекой pandas, этот синтаксис покажется вам до боли знакомым, и вы можете поддаться соблазну использовать подобные конструкции в Polars.

И хотя этот код выдает такой же результат, как и предыдущий, выполняется он далеко не оптимально. В частности, условия вычисляются до применения метода `df.filter()`. А поскольку выражения здесь не используются, оптимизатор никак не может вмешаться в процесс. Кроме того, указанные критерии вычисляются последовательно, а не параллельно.

Это становится еще более очевидно, когда мы применяем несколько операций к ленивому датафрейму. Сейчас очень внимательно следите за руками, потому что это очень важно! Взгляните на пример сложной фильтрации с использованием выражений:

```
(
    fruit.lazy()
    .filter((pl.col("weight") > 1000) & pl.col("is_round"))
    .with_columns(is_berry=pl.col("name").str.ends_with("berry"))
    .collect()
)
```

Вывод:

```
shape: (2, 6)
┌────────────┬────────┬────────┬──────────┬────────┬──────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin ┆ is_berry │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---    ┆ ---      │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str    ┆ bool     │
╞════════════╪════════╪════════╪══════════╪════════╪══════════╡
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa ┆ false    │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa ┆ false    │
└────────────┴────────┴────────┴──────────┴────────┴──────────┘
```

А теперь посмотрим на пример без использования выражений:

```
(
    fruit.lazy()
    .filter((fruit["weight"] > 1000) & fruit["is_round"])
    .with_columns(is_berry=fruit["name"].str.ends_with("berry"))
    .collect()
)
```

Ответьте с одного раза, успешно ли он выполнится? И...

Вывод:

```
ShapeError: unable to add a column of length 10 to a DataFrame of height 2
```

Дословно в ошибке говорится о невозможности добавить столбец длины 10 к датафрейму всего с двумя строками. Все правильно: в отсутствие выражений Polars не имеет возможности использовать оптимизации, в результате чего вся ответственность за порядок выполнения операций ложится на вас

как на разработчика. В данном случае причина ошибки состоит в том, что метод `df.filter()` сократил количество строк в датафрейме до двух, тогда как переменная `fruit` по-прежнему указывает на датафрейм с десятью строками.

Именно поэтому мы всегда рекомендуем вам использовать выражения. В Polars быть ленивым не зазорно.

# Заключение

В этой главе мы познакомились с выражениями в Polars и научились их использовать. Вы узнали, что:

- выражения в Polars представляют собой повторно используемые ленивые строительные блоки, позволяющие эффективно манипулировать данными, включая отбор, фильтрацию и создание новых столбцов;
- выражения обладают достаточной гибкостью и позволяют эффективно выполнять арифметические операции, операции сравнения и манипуляции со строками, что делает их в высшей степени универсальными;
- создавать выражения можно на основе существующих столбцов, литеральных значений или диапазонов, что позволяет использовать методы для преобразования и управления данными;
- Polars оптимизирует выражения и вычисляет их параллельно, что значительно повышает эффективность обработки данных;
- переименование выражений можно использовать с целью облегчения восприятия данных и во избежание дублирования имен столбцов. Для этого можно воспользоваться методом `Expr.alias()` или ключевыми параметрами;
- настоятельно рекомендуется отдавать предпочтение использованию выражений в сравнении с выполнением операций с объектами Series напрямую с целью повышения быстродействия и осуществления параллельных вычислений.

В следующей главе мы узнаем, как можно эффективно расширять и комбинировать выражения для выполнения сложных операций над данными.

# Глава 8

# Продолжаем знакомиться с выражениями

В предыдущей главе мы узнали, как можно создавать выражения с нуля. Но на одних таких выражениях далеко не уедешь. В этой главе мы продолжим знакомиться с выражениями и научимся добавлять к ним дополнительные операции (или методы).

Вы узнаете, как при помощи выражений:

- выполнять математические преобразования;
- работать с пропущенными значениями;
- применять операции сглаживания;
- выбирать конкретные значения;
- агрегировать значения с использованием статистики.

**Разнообразие методов**

В этой главе мы упомянем более 138 методов для работы с выражениями. Разумеется, мы не сможем продемонстрировать на практике их все. Детально со всеми методами вы можете ознакомиться в официальной документации Polars по адресу *https://docs.pola.rs/api/python/stable/reference/index.html*.

В некоторых фрагментах кода из этой главы мы будем использовать модули *math* и *numpy* для доступа к математическим константам, таким как `math.pi`, и генерирования случайных последовательностей:

```
import math
import numpy as np

print(f"{math.pi=}")
rng = np.random.default_rng(1729)
print(f"{rng.random()=}")
```

Вывод:

```
math.pi=3.141592653589793
rng.random()=0.03074202960516803
```

Давайте познакомимся со структурой этой главы.

Все инструкции по загрузке и установке требуемых файлов вы найдете в главе 2. Мы будем исходить из предположения о том, что все рабочие файлы вы поместили в директорию *data*.

# Типы операций

Вместо того чтобы обрушивать на вас лавину из 138 методов для работы с выражениями одним огромным списком, мы решили разбить их на пять групп по типу входа и форме вывода. В рамках этих групп мы объединили методы в категории. Методы, не вошедшие ни в одну категорию, мы поместили в раздел *Прочие операции*. На рис. 8.1 показаны образованные в результате группы операций для работы с выражениями.

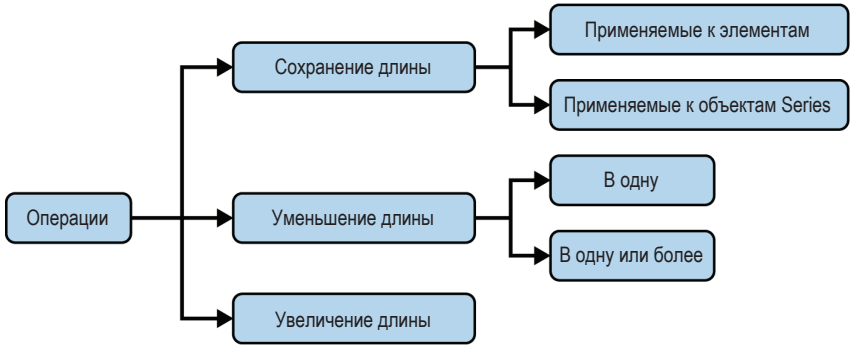



**Рис. 8.1** ❖ Типы операций для работы с выражениями

**Схожие методы – разные разделы**

Хотя мы уверены в правильности выбора организации главы, некоторые схожие методы в результате оказались разнесены по разным разделам. К примеру, оба метода `Expr.unique()` и `Expr.is_unique()` предназначены для работы с уникальными значениями, но поскольку первый из них может сокращать длину объекта Series, а второй – нет, они оказались в разных категориях.

Ниже мы приведем четыре примера, чтобы вы лучше поняли, что мы имеем в виду, говоря о типах операций.

# Пример A: поэлементные операции

В первом примере мы воспользуемся двумя методами для создания дополнительных столбцов: *`Expr.sqrt()`* и *`Expr.exp()`*. Оба метода работают поэле-

ментно, т. е. обрабатывают за раз только один элемент, и в результате их применения длина объекта Series остается неизменной:

```
penguins = pl.read_csv("data/penguins.csv", null_values="NA").select(
    "species",
    "island",
    "sex",
    "year",
    mass=pl.col("body_mass_g") / 1000,
)
penguins.with_columns(
    mass_sqrt=pl.col("mass").sqrt(), ❶
    mass_exp=pl.col("mass").exp(),
)
```

❶ Метод `Expr.sqrt()` вычисляет квадратный корень из значений, находящихся в столбце `mass`. Обратите внимание, что значения `null` затронуты не были.

Вывод:

```
shape: (344, 7)
```

| species | island | sex | year | mass | mass_sqrt | mass_exp |
|---|---|---|---|---|---|---|
| str | str | str | i64 | f64 | f64 | f64 |
| Adelie | Torgersen | male | 2007 | 3.75 | 1.936492 | 42.521082 |
| Adelie | Torgersen | female | 2007 | 3.8 | 1.949359 | 44.701184 |
| Adelie | Torgersen | female | 2007 | 3.25 | 1.802776 | 25.79034 |
| Adelie | Torgersen | null | 2007 | null | null | null |
| Adelie | Torgersen | female | 2007 | 3.45 | 1.857418 | 31.500392 |
| … | … | … | … | … | … | … |
| Chinstrap | Dream | male | 2009 | 4.0 | 2.0 | 54.59815 |
| Chinstrap | Dream | female | 2009 | 3.4 | 1.843909 | 29.9641 |
| Chinstrap | Dream | male | 2009 | 3.775 | 1.942936 | 43.597508 |
| Chinstrap | Dream | male | 2009 | 4.1 | 2.024846 | 60.340288 |
| Chinstrap | Dream | female | 2009 | 3.775 | 1.942936 | 43.597508 |

Теперь посмотрим на пример, в котором несколько исходных строк сводятся в одну.

## Пример B: операции агрегации в одну строку

Во втором примере мы применим два метода, сводящих объект Series к единственной строке (по столбцу `mass` мы получили арифметическое среднее, а из столбца `island` взяли первое значение):

```
penguins.select(pl.col("mass").mean(), pl.col("island").first())
```

Вывод:

```
shape: (1, 2)
┌──────────┬────────────┐
│ mass     ┆ island     │
│ ---      ┆ ---        │
│ f64      ┆ str        │
╞══════════╪════════════╡
│ 4.201754 ┆ Torgersen  │
└──────────┴────────────┘
```

Взглянув на эти методы, посмотрим на пример, в котором количество строк может сокращаться не до одной, а до одной или нескольких.

## Пример C: операции агрегации в одну или несколько строк

В данном случае мы воспользуемся методом *Expr.unique()* для извлечения уникальных значений из объекта Series. Выполнение такого типа операций приводит к образованию одного или нескольких значений, поскольку уникальных значений может быть сколько угодно:

```
penguins.select(pl.col("island").unique())
```

Вывод:

```
shape: (3, 1)
┌────────────┐
│ island     │
│ ---        │
│ str        │
╞════════════╡
│ Torgersen  │
│ Biscoe     │
│ Dream      │
└────────────┘
```

Теперь, когда мы рассмотрели все варианты снижения размерности исходного объекта, рассмотрим пример ее расширения.

## Пример D: операции, расширяющие исходный объект

Здесь мы воспользуемся методом *Expr.extend_constant()* для добавления заданных значений в конец объекта Series. Такой тип операций встречается гораздо реже в сравнении с операциями снижения размерности. Этот пример

выглядит несколько надуманным, но он показывает, насколько мощными могут быть выражения при добавлении дополнительных методов:

```
penguins.select(
    pl.col("species")
    .unique() ❶
    .repeat_by(3000) ❷
    .explode() ❸
    .extend_constant("Saiyan", n=1) ❹
)
```

❶ Получаем три уникальных значения из объекта Series.
❷ Повторяем каждое значение 3000 раз. В результате получаем объект Series, состоящий из трех длинных списков.
❸ Вызываем метод `explode()` для получения одного длинного объекта Series, состоящего из 9000 значений.
❹ Добавляем одно значение в конец Series. В результате получим объект Series длины 9001, как показано ниже:

```
shape: (9_001, 1)
┌─────────────┐
│ species     │
│ ---         │
│ str         │
╞═════════════╡
│ Adelie      │
│ Adelie      │
│ Adelie      │
│ Adelie      │
│ Adelie      │
│ …           │
│ Chinstrap   │
│ Chinstrap   │
│ Chinstrap   │
│ Chinstrap   │
│ Saiyan      │
└─────────────┘
```

Эти четыре примера должны помочь вам получить общее представление о типах операций, которые можно применять совместно с выражениями.

# Поэлементные операции

В этом разделе мы поговорим об операциях, применяющихся за раз к одному элементу объекта Series. Каждое значение вычисляется независимо, и порядок, в котором они располагаются, не имеет значения. В качестве примера такой операции мы уже рассмотрели извлечение квадратного корня из всех значений в объекте Series с помощью метода `Expr.sqrt()`. Также можно упомянуть метод `Expr.round()`, предназначенный для округления значений в объекте Series.

В следующих пяти разделах мы рассмотрим поэлементные операции для выполнения математических преобразований, тригонометрических операций, округления, разбиения на категории, работы с пропущенными или бесконечными значениями и многого другого.

## Операции для выполнения математических преобразований

Математические операции, такие как взятие логарифма или извлечение квадратного корня, формируют основу любых задач, связанных с данными. В табл. 8.1 мы перечислили методы выражений для реализации таких преобразований. Арифметические операции между двумя выражениями, такие как сложение и умножение, мы рассмотрим в главе 9, поскольку они касаются комбинирования выражений.

***Таблица 8.1. Поэлементные операции для выполнения простых математических преобразований***

| Метод | Описание |
|---|---|
| *Expr.abs()* | Вычисляет значение по модулю |
| *Expr.cbrt()* | Вычисляет кубический корень из элементов |
| *Expr.exp()* | Поэлементно вычисляет экспоненту |
| *Expr.log()* | Вычисляет логарифм с произвольным основанием |
| *Expr.log10()* | Поэлементно вычисляет логарифм с основанием 10 |
| *Expr.log1p()* | Поэлементно вычисляет натуральный логарифм с прибавлением единицы |
| *Expr.sign()* | Поэлементно определяет знак числа |
| *Expr.sqrt()* | Вычисляет квадратный корень из элементов |

Применение методов `Expr.abs()`, `Expr.exp()`, `Expr.log()`, `Expr.log10()`, `Expr.log1p()`, `Expr.sign()` и `Expr.sqrt()` демонстрируется в следующем фрагменте кода. Метод `Expr.cbrt()` ничем от них не отличается:

```
(
    pl.DataFrame({"x": [-2.0, 0.0, 0.5, 1.0, math.e, 1000.0]}).with_columns(
        abs=pl.col("x").abs(),
        exp=pl.col("x").exp(),
        log2=pl.col("x").log(2), ❶
        log10=pl.col("x").log10(),
        log1p=pl.col("x").log1p(),
        sign=pl.col("x").sign(),
        sqrt=pl.col("x").sqrt(),
    )
)
```

❶ Метод `Expr.log()` является единственным из перечисленных, который может принимать опциональный аргумент, а именно основание для вычисления логарифма.

По умолчанию вычисляется натуральный логарифм.

Вывод:

```
shape: (6, 8)
```

| x | abs | exp | log2 | log10 | log1p | sign | sqrt |
|---|---|---|---|---|---|---|---|
| f64 | f64 | f64 | f64 | f64 | f64 | f64 | f64 |
| -2.000 | 2.000 | 0.135 | NaN | NaN | NaN | -1.000 | NaN |
| 0.000 | 0.000 | 1.000 | -inf | -inf | 0.000 | 0.000 | 0.000 |
| 0.500 | 0.500 | 1.649 | -1.000 | -0.301 | 0.405 | 1.000 | 0.707 |
| 1.000 | 1.000 | 2.718 | 0.000 | 0.000 | 0.693 | 1.000 | 1.000 |
| 2.718 | 2.718 | 15.154 | 1.443 | 0.434 | 1.313 | 1.000 | 1.649 |
| 1000.000 | 1000.000 | inf | 9.966 | 3.000 | 6.909 | 1.000 | 31.623 |

# Тригонометрические операции

Тригонометрия – это раздел математики, изучающий соотношения между углами и длинами сторон в треугольниках. Она довольно часто используется в различных областях науки о данных, включая обработку сигналов, анализ специфических данных и конструирование признаков. В табл. 8.2 приведены методы выражений для работы с тригонометрией в Polars.

***Таблица 8.2. Поэлементные методы для выполнения тригонометрических операций***

| Метод | Описание |
|---|---|
| *Expr.arccos()* | Поэлементно вычисляет обратный косинус |
| *Expr.arccosh()* | Поэлементно вычисляет обратный гиперболический косинус |
| *Expr.arcsin()* | Поэлементно вычисляет обратный синус |
| *Expr.arcsinh()* | Поэлементно вычисляет обратный гиперболический синус |
| *Expr.arctan()* | Поэлементно вычисляет обратный тангенс |
| *Expr.arctanh()* | Поэлементно вычисляет обратный гиперболический тангенс |
| *Expr.cos()* | Поэлементно вычисляет косинус |
| *Expr.cosh()* | Поэлементно вычисляет гиперболический косинус |
| *Expr.degrees()* | Преобразовывает радианы в градусы |
| *Expr.radians()* | Преобразовывает градусы в радианы |
| *Expr.sin()* | Поэлементно вычисляет синус |
| *Expr.sinh()* | Поэлементно вычисляет гиперболический синус |
| *Expr.tan()* | Поэлементно вычисляет тангенс |
| *Expr.tanh()* | Поэлементно вычисляет гиперболический тангенс |

В приведенном ниже фрагменте кода мы рассмотрим применение методов `Expr.arccos()`, `Expr.cos()`, `Expr.degrees()`, `Expr.radians()` и `Expr.sin()`. Оставшиеся методы, а именно `Expr.arccosh()`, `Expr.arcsin()`, `Expr.arc sinh()`, `Expr.`

`arctan()`, `Expr.arctanh()`, `Expr.cosh()`, `Expr.sinh()`, `Expr.tan()` и `Expr.tanh()`, можно применять аналогичным образом. Ни один из этих методов не требует передачи аргументов:

```
(
    pl.DataFrame(
        {"x": [-math.pi, 0.0, 1.0, math.pi, 2 * math.pi, 90.0, 180.0, 360.0]}
    ).with_columns(
        arccos=pl.col("x").arccos(), ❶
        cos=pl.col("x").cos(),
        degrees=pl.col("x").degrees(),
        radians=pl.col("x").radians(),
        sin=pl.col("x").sin(),
    )
)
```

❶ При получении в результате выполнения поэлементной операции значения `NaN` другие значения не затрагиваются.

Вывод:

```
shape: (8, 6)
```

| x | arccos | cos | degrees | radians | sin |
|---|---|---|---|---|---|
| f64 | f64 | f64 | f64 | f64 | f64 |
| -3.141593 | NaN | -1.0 | -180.0 | -0.054831 | -1.2246e-16 |
| 0.0 | 1.570796 | 1.0 | 0.0 | 0.0 | 0.0 |
| 1.0 | 0.0 | 0.540302 | 57.29578 | 0.017453 | 0.841471 |
| 3.141593 | NaN | -1.0 | 180.0 | 0.054831 | 1.2246e-16 |
| 6.283185 | NaN | 1.0 | 360.0 | 0.109662 | -2.4493e-16 |
| 90.0 | NaN | -0.448074 | 5156.620156 | 1.570796 | 0.893997 |
| 180.0 | NaN | -0.59846 | 10313.240312 | 3.141593 | -0.801153 |
| 360.0 | NaN | -0.283691 | 20626.480625 | 6.283185 | 0.958916 |

## Операции для округления и разбиения на категории

Бывает так, что в ваших данных наблюдается избыток точности или чересчур большое количество уникальных значений. В таких случаях бывает полезно округлить значения или разбить их на дискретные категории. В табл. 8.3 показаны методы в Polars, предназначенные как раз для этого[1].

[1] Чисто технически метод `Expr.qcut()` не относится к поэлементным, поскольку вычисление процентилей основывается на всем объекте Series. Но мы решили, что для удобства будет лучше разместить этот метод рядом со своим братом – методом `Expr.cut()`.

***Таблица 8.3. Поэлементные методы для округления и разбиения на категории***

| Метод | Описание |
|---|---|
| *Expr.ceil()* | Округляет значения вверх до ближайшего целого |
| *Expr.clip()* | Ограничивает значения в массиве с использованием верхней и нижней границ |
| *Expr.cut()* | Разбивает непрерывные значения на дискретные категории |
| *Expr.floor()* | Округляет значения вниз до ближайшего целого |
| *Expr.qcut()* | Разбивает непрерывные значения на дискретные категории на основе квантилей |
| *Expr.round()* | Округляет значения до заданного количества знаков после запятой |

В примере ниже мы продемонстрируем работу представленных в табл. 8.3 методов применительно к диапазону чисел, а методом `Expr.round()` воспользуемся дважды:

```
(
    pl.DataFrame(
        {"x": [-6.0, -0.5, 0.0, 0.5, math.pi, 9.9, 9.99, 9.999]}
    ).with_columns(
        ceil=pl.col("x").ceil(),
        clip=pl.col("x").clip(-1, 1),
        cut=pl.col("x").cut([-1, 1], labels=["bad", "neutral", "good"]), ❶
        floor=pl.col("x").floor(),
        qcut=pl.col("x").qcut([0.5], labels=["below median", "above median"]),
        round2=pl.col("x").round(2),
        round0=pl.col("x").round(0), ❷
    )
)
```

❶ В результате применения методов `Expr.cut()` и `Expr.qcut()` создается категориальный объект Series. Если вы хотите, чтобы он стал целочисленным, можете добавить к выражению, например, метод преобразования типов `Expr.cast(pl.Int64)`.

❷ Даже при округлении чисел до полного отсутствия знаков после запятой (к примеру, с помощью методов `Expr.round(0)`, `Expr.ceil()` или `Expr.floor()`) тип объекта Series останется вещественным.

Вывод:

```
shape: (8, 8)
┌──────────┬──────┬──────┬─────────┬───────┬──────────────┬────────┬────────┐
│ x        ┆ ceil ┆ clip ┆ cut     ┆ floor ┆ qcut         ┆ round2 ┆ round0 │
│ ---      ┆ ---  ┆ ---  ┆ ---     ┆ ---   ┆ ---          ┆ ---    ┆ ---    │
│ f64      ┆ f64  ┆ f64  ┆ cat     ┆ f64   ┆ cat          ┆ f64    ┆ f64    │
╞══════════╪══════╪══════╪═════════╪═══════╪══════════════╪════════╪════════╡
│ -6.0     ┆ -6.0 ┆ -1.0 ┆ bad     ┆ -6.0  ┆ below median ┆ -6.0   ┆ -6.0   │
│ -0.5     ┆ -0.0 ┆ -0.5 ┆ neutral ┆ -1.0  ┆ below median ┆ -0.5   ┆ -1.0   │
│ 0.0      ┆ 0.0  ┆ 0.0  ┆ neutral ┆ 0.0   ┆ below median ┆ 0.0    ┆ 0.0    │
│ 0.5      ┆ 1.0  ┆ 0.5  ┆ neutral ┆ 0.0   ┆ below median ┆ 0.5    ┆ 1.0    │
│ 3.141593 ┆ 4.0  ┆ 1.0  ┆ good    ┆ 3.0   ┆ above median ┆ 3.14   ┆ 3.0    │
│ 9.9      ┆ 10.0 ┆ 1.0  ┆ good    ┆ 9.0   ┆ above median ┆ 9.9    ┆ 10.0   │
```

```
| 9.99     | 10.0 | 1.0  | good    | 9.0   | above median | 9.99   | 10.0   |
| 9.999    | 10.0 | 1.0  | good    | 9.0   | above median | 10.0   | 10.0   |
└──────────┴──────┴──────┴─────────┴───────┴──────────────┴────────┴────────┘
```

## Операции для работы с пропущенными или бесконечными значениями

В реальных наборах данных очень часто присутствуют пропущенные значения. Значения `NaN` или бесконечные значения могут появляться в исходных данных в результате выполнения некорректных преобразований. В Polars представлено несколько удобных методов для работы с подобными значениями, которые перечислены в табл. 8.4. Позже в этой главе мы увидим еще несколько методов для работы с пропущенными значениями, которые взаимодействуют с целыми объектами Series.

***Таблица 8.4. Поэлементные методы для работы с пропущенными или бесконечными значениями***

| Метод | Описание |
|---|---|
| *Expr.fill_nan()* | Заполняет значения `NaN` переданными значениями |
| *Expr.fill_null()* | Заполняет значения `null` переданными значениями или в соответствии с выбранной стратегией |
| *Expr.is_finite()* | Возвращает булев объект Series с индикаторами того, являются ли значения конечными |
| *Expr.is_infinite()* | Возвращает булев объект Series с индикаторами того, являются ли значения бесконечными |
| *Expr.is_nan()* | Возвращает булев объект Series с индикаторами того, являются ли элементы значениями `NaN` |
| *Expr.is_not_nan()* | Возвращает булев объект Series с индикаторами того, не являются ли элементы значениями `NaN` |
| *Expr.is_not_null()* | Возвращает булев объект Series с индикаторами того, не являются ли элементы значениями `null` |
| *Expr.is_null()* | Возвращает булев объект Series с индикаторами того, являются ли элементы значениями `null` |

В следующем фрагменте кода мы покажем, как можно использовать на практике методы `Expr.fill_nan()`, `Expr.fill_null()`, `Expr.is_finite()`, `Expr.is_infinite()`, `Expr.is_nan()` и `Expr.is_null()` применительно к ряду числовых значений с пропусками и бесконечными значениями. Методы `Expr.is_not_nan()` и `Expr.is_not_null()` производят результаты, обратные методам `Expr.is_nan()` и `Expr.is_null()` соответственно.

```
x = [42.0, math.nan, None, math.inf, -math.inf]
(
    pl.DataFrame({"x": x}).with_columns(
        fill_nan=pl.col("x").fill_nan(999),
        fill_null=pl.col("x").fill_null(0), ❶
```

```
        is_finite=pl.col("x").is_finite(),
        is_infinite=pl.col("x").is_infinite(),
        is_nan=pl.col("x").is_nan(),
        is_null=pl.col("x").is_null(),
    )
)
```

❶ Метод `Expr.fill_null()` располагает ключевым аргументом `strategy`, позволяющим усложнить процесс заполнения пропущенных значений путем учета соседствующих значений. Мы уже говорили об этом в главе 4.

Вывод:

```
shape: (5, 7)
```

| x | fill_nan | fill_null | is_finite | is_infinite | is_nan | is_null |
|---|---|---|---|---|---|---|
| f64 | f64 | f64 | bool | bool | bool | bool |
| 42.0 | 42.0 | 42.0 | true | false | false | false |
| NaN | 999.0 | NaN | false | false | true | false |
| null | null | 0.0 | null | null | null | true |
| inf | inf | inf | false | true | false | false |
| -inf | -inf | -inf | false | true | false | false |

**NaN против null**

Всегда стоит помнить, что *NaN* и *null* – это далеко не одно и то же. Если вам необходимо заменить в объекте Series и значения `NaN`, и значения `null`, вы можете воспользоваться методами `Expr.fill_nan()` и `Expr.fill_null()`. А если вам нужно узнать, является ли значение `Nan` или `null`, можно скомбинировать вызов методов `Expr.is_nan()` и `Expr.is_null()` при помощи булева оператора OR (*|*):

```
(
    pl.DataFrame({"x": x}).with_columns(
        fill_both=pl.col("x").fill_nan(0).fill_null(0),
        is_either=(pl.col("x").is_nan() |
        pl.col("x").is_null()),
    )
)
```

Вывод:

```
shape: (5, 3)
```

| x | fill_both | is_either |
|---|---|---|
| f64 | f64 | bool |
| 42.0 | 42.0 | false |
| NaN | 0.0 | true |
| null | 0.0 | true |
| inf | inf | false |

```
| -inf | -inf      | false     |
└──────┴───────────┴───────────┘
```

Подробнее о булевых операторах мы будем говорить в следующей главе. Обрабатывать ли значения `NaN` и `null` одинаково или по-разному, зависит от конкретной задачи.

# Прочие операции

Есть еще три поэлементных метода, не вошедших ни в одну из перечисленных выше категорий. Мы перечислили их в табл. 8.5.

***Таблица 8.5. Прочие поэлементные методы***

| Метод | Описание |
|---|---|
| *`Expr.hash()`* | Генерирует хеш для элементов в выборке |
| *`Expr.repeat_by()`* | Повторяет элементы в объекте Series |
| *`Expr.replace()`* | Заменяет значения в столбце в соответствии с переданным словарем |

В следующем фрагменте кода показан пример применения всех трех методов:

```
(
    pl.DataFrame({"x": ["here", "there", "their", "they're"]}).with_columns(
        hash=pl.col("x").hash(seed=1337), ❶
        repeat_by=pl.col("x").repeat_by(3),
        replace=pl.col("x").replace(
            {
                "here": "there",
                "they're": "they are",
            }
        ),
    )
)
```

❶ Метод `Expr.hash()` на разных компьютерах и версиях библиотеки Polars может генерировать разные значения хеша. Подробнее можно почитать на странице *https://docs.rs/ahash/latest/ahash*.

Вывод:

```
shape: (4, 4)
┌─────────┬──────────────────────┬──────────────────────────────┬──────────┐
│ x       ┆ hash                 ┆ repeat_by                    ┆ replace  │
│ ---     ┆ ---                  ┆ ---                          ┆ ---      │
│ str     ┆ u64                  ┆ list[str]                    ┆ str      │
╞═════════╪══════════════════════╪══════════════════════════════╪══════════╡
│ here    ┆ 12695211751326448172 ┆ ["here", "here", "here"]     ┆ there    │
│ there   ┆ 17327794691236705436 ┆ ["there", "there", "there"]  ┆ there    │
│ their   ┆ 2663095961041830581  ┆ ["their", "their", "their"]  ┆ their    │
│ they're ┆ 6743063676290245144  ┆ ["they're", "they're", "they'r… ┆ they are │
└─────────┴──────────────────────┴──────────────────────────────┴──────────┘
```

# Не снижающие размерность операции, применяющиеся к объектам Series

В оставшихся разделах этой главы мы переключимся с поэлементных операций на операции, применяющиеся к целым объектам Series. В результате их выполнения объекты Series будут меняться целиком, а сами значения (а иногда и их порядок) будут зависеть друг от друга. В качестве примеров таких операций можно назвать метод `Expr.cum_sum()`, применяющийся для вычисления накопительной суммы значений, и метод `Expr.forward_fill()`, использующийся для замены пропущенных значений прямым проходом.

В следующих шести разделах мы рассмотрим операции, не ведущие к изменению количества строк в объекте Series, включая накопление, заполнение, сдвиг, расчет скользящих показателей, сортировку и многое другое.

## Накопительные операции

Накопительные операции проходят по объекту Series и наращивают, к примеру, сумму или максимальное значение. В табл. 8.6 показаны имеющиеся в Polars накопительные методы.

***Таблица 8.6. Накопительные методы***

| Метод | Описание |
|---|---|
| *`Expr.cum_count()`* | Возвращает массив с накопленными значениями количества в объекте Series |
| *`Expr.cum_max()`* | Возвращает массив с накопленными максимальными значениями в объекте Series |
| *`Expr.cum_min()`* | Возвращает массив с накопленными минимальными значениями в объекте Series |
| *`Expr.cum_prod()`* | Возвращает массив с накопленными произведениями в объекте Series |
| *`Expr.cum_sum()`* | Возвращает массив с накопленными суммами в объекте Series |
| *`Expr.diff()`* | Рассчитывает разницу между значениями с заданным шагом |
| *`Expr.pct_change()`* | Рассчитывает разницу в процентах между значениями с заданным шагом |

Все эти методы на вход принимают аргумент `reverse`, который говорит о том, необходимо ли развернуть объект Series перед применением операции. В следующем фрагменте кода мы применим все эти методы к числовому ряду с пропусками и значениями `NaN`:

```
(
    pl.DataFrame(
        {"x": [0.0, 1.0, 2.0, None, 2.0, np.nan, -1.0, 2.0]}
    ).with_columns(
```

```
        cum_count=pl.col("x").cum_count(), ❶
        cum_max=pl.col("x").cum_max(),
        cum_min=pl.col("x").cum_min(),
        cum_prod=pl.col("x").cum_prod(reverse=True), ❷
        cum_sum=pl.col("x").cum_sum(),
        diff=pl.col("x").diff(),
        pct_change=pl.col("x").pct_change(),
    )
)
```

❶ Метод `Expr.cum_count()` не учитывает пропущенные значения.
❷ Если бы мы предварительно не развернули числовой ряд, результирующий столбец оказался бы заполнен нулями.

Вывод:

```
shape: (8, 8)
```

| x | cum_count | cum_max | cum_min | cum_prod | cum_sum | diff | pct_change |
|---|---|---|---|---|---|---|---|
| f64 | u32 | f64 | f64 | f64 | f64 | f64 | f64 |
| 0.0 | 1 | 0.0 | 0.0 | NaN | 0.0 | null | null |
| 1.0 | 2 | 1.0 | 0.0 | NaN | 1.0 | 1.0 | inf |
| 2.0 | 3 | 2.0 | 0.0 | NaN | 3.0 | 1.0 | 1.0 |
| null | 3 | null | null | null | null | null | 0.0 |
| 2.0 | 4 | 2.0 | 0.0 | NaN | 5.0 | null | 0.0 |
| NaN | 5 | 2.0 | 0.0 | NaN | NaN | NaN | NaN |
| -1.0 | 6 | 2.0 | -1.0 | -2.0 | NaN | NaN | NaN |
| 2.0 | 7 | 2.0 | -1.0 | 2.0 | NaN | 3.0 | -3.0 |

**Заразные значения NaN**

Значения `NaN` могут оказывать существенное влияние на вывод операций, применяющихся к объектам Series. В приведенном выше примере:

- вывод методов `Expr.cum_count()`, `Expr.cum_max()` и `Expr.cum_min()` не пострадал;
- вывод методов `Expr.cum_prod()` и `Expr.cum_sum()` «заразился» значениями `NaN` при первой встрече с ними;
- вывод методов `Expr.diff()` и `Expr.pct_change()` «заразился» значениями `NaN` только на две следующие строки после встречи.

## Операции заполнения и смещения

Часто бывает полезно обрабатывать данные в строках с использованием значений в их окрестностях. К примеру, вам может понадобиться сдвинуть значения на определенное количество строк или заполнить пропуски в данных на основе строк, предшествующих или следующих за текущей строкой. В таких случаях вам могут очень помочь операции заполнения и смещения.

В табл. 8.7 приведены такие операции, применяющиеся к объектам Series.

***Таблица 8.7. Методы заполнения и смещения***

| Метод | Описание |
|---|---|
| *Expr.backward_fill()* | Заполняет пропущенные значения обратным проходом – с использованием ближайшего следующего непропущенного значения |
| *Expr.forward_fill()* | Заполняет пропущенные значения прямым проходом – с использованием ближайшего предыдущего непропущенного значения |
| *Expr.interpolate()* | Заполняет пропущенные значения при помощи интерполяции |
| *Expr.shift()* | Сдвигает значения на заданное количество строк |

Давайте применим методы `Expr.backward_fill()`, `Expr.forward_fill()`, `Expr.interpolate()` (дважды) и `Expr.shift()` (дважды) для числового ряда с пропущенными значениями:

```
(
    pl.DataFrame(
        {"x": [-1.0, 0.0, 1.0, None, None, 3.0, 4.0, math.nan, 6.0]}
    ).with_columns(
        backward_fill=pl.col("x").backward_fill(), ❶
        forward_fill=pl.col("x").forward_fill(limit=1),
        interp1=pl.col("x").interpolate(method="linear"), ❷
        interp2=pl.col("x").interpolate(method="nearest"),
        shift1=pl.col("x").shift(1),
        shift2=pl.col("x").shift(-2),
    )
)
```

❶ Значения `NaN` не заполняются и не интерполируются.

❷ Обратите внимание на разницу между двумя стратегиями интерполяции: `linear` и `nearest`. В первом случае интерполяция производится в интервале от ближайшего предыдущего до ближайшего следующего непропущенного значения, а во втором берется просто ближайшее непропущенное значение (ближайшее предыдущее или ближайшее следующее в зависимости от того, какое ближе).

Вывод:

```
shape: (9, 7)
```

| x | backward_fill | forward_fill | interp1 | interp2 | shift1 | shift2 |
|---|---|---|---|---|---|---|
| f64 | f64 | f64 | f64 | f64 | f64 | f64 |
| -1.0 | -1.0 | -1.0 | -1.0 | -1.0 | null | 1.0 |
| 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | -1.0 | null |
| 1.0 | 1.0 | 1.0 | 1.0 | 1.0 | 0.0 | null |
| null | 3.0 | 1.0 | 1.666667 | 1.0 | 1.0 | 3.0 |
| null | 3.0 | null | 2.333333 | 3.0 | null | 4.0 |
| 3.0 | 3.0 | 3.0 | 3.0 | 3.0 | null | NaN |
| 4.0 | 4.0 | 4.0 | 4.0 | 4.0 | 3.0 | 6.0 |
| NaN | NaN | NaN | NaN | NaN | 4.0 | null |
| 6.0 | 6.0 | 6.0 | 6.0 | 6.0 | NaN | null |

# Операции, связанные с дублирующимися значениями

Существует четыре не снижающих размерность операции, применяющихся к объектам Series, которые призваны облегчить работу с уникальными и дублирующимися значениями. Все они приведены в табл. 8.8. Есть и другие методы, схожие с этими, но поскольку они уменьшают размерность объекта Series, мы будем говорить о них далее.

***Таблица 8.8. Методы для работы с дублирующимися значениями***

| Метод | Описание |
|---|---|
| *Expr.is_duplicated()* | Возвращает булев объект Series с индикаторами того, являются ли элементы дубликатами |
| *Expr.is_first_distinct()* | Возвращает булев объект Series с индикаторами того, являются ли элементы первыми в ряду дублирующихся значений |
| *Expr.is_last_distinct()* | Возвращает булев объект Series с индикаторами того, являются ли элементы последними в ряду дублирующихся значений |
| *Expr.is_unique()* | Возвращает булев объект Series с индикаторами того, являются ли элементы уникальными |

Применим все четыре метода к столбцу со строковым содержимым:

```
(
    pl.DataFrame({"x": ["A", "C", "D", "C"]}).with_columns( ❶
        is_duplicated=pl.col("x").is_duplicated(),
        is_first_distinct=pl.col("x").is_first_distinct(),
        is_last_distinct=pl.col("x").is_last_distinct(),
        is_unique=pl.col("x").is_unique(),
    )
)
```

❶ Помните, что многие из подобных методов могут применяться и к значениям других типов данных.

Вывод:

```
shape: (4, 5)
┌─────┬──────────────┬────────────────┬───────────────┬───────────┐
│ x   ┆ is_duplicated ┆ is_first_distinct ┆ is_last_distinct ┆ is_unique │
│ --- ┆ ---          ┆ ---            ┆ ---           ┆ ---       │
│ str ┆ bool         ┆ bool           ┆ bool          ┆ bool      │
╞═════╪══════════════╪════════════════╪═══════════════╪═══════════╡
│ A   ┆ false        ┆ true           ┆ true          ┆ true      │
│ C   ┆ true         ┆ true           ┆ false         ┆ false     │
│ D   ┆ false        ┆ true           ┆ true          ┆ true      │
│ C   ┆ true         ┆ false          ┆ true          ┆ false     │
└─────┴──────────────┴────────────────┴───────────────┴───────────┘
```

# Операции для расчета скользящих показателей

В анализе данных часто применяются скользящие показатели, позволяющие сгладить мелкодисперсные колебания в данных. Далее мы посмотрим пример использования техники сглаживания при анализе цен на акции.

Методы для расчета скользящих показателей приведены в табл. 8.9.

***Таблица 8.9. Методы для расчета скользящих показателей***

| Метод | Описание |
|---|---|
| *Expr.ewm_mean()* | Вычисляет экспоненциально-взвешенное скользящее среднее |
| *Expr.ewm_std()* | Вычисляет экспоненциально-взвешенное скользящее стандартное отклонение |
| *Expr.ewm_var()* | Вычисляет экспоненциально-взвешенную скользящую дисперсию |
| *Expr.rolling_apply()* | Применяет пользовательскую скользящую функцию |
| *Expr.rolling_map()* | Вычисляет пользовательскую скользящую функцию |
| *Expr.rolling_max()* | Применяет функцию скользящего максимума к значениям в массиве |
| *Expr.rolling_mean()* | Применяет функцию скользящего среднего к значениям в массиве |
| *Expr.rolling_median()* | Вычисляет скользящую медиану |
| *Expr.rolling_min()* | Применяет функцию скользящего минимума к значениям в массиве |
| *Expr.rolling_quantile()* | Применяет функцию скользящего квантиля к значениям в массиве |
| *Expr.rolling_skew()* | Вычисляет скользящий показатель перекоса (асимметрии) распределения |
| *Expr.rolling_std()* | Вычисляет скользящее стандартное отклонение |
| *Expr.rolling_sum()* | Применяет функцию скользящей суммы к значениям в массиве |
| *Expr.rolling_var()* | Вычисляет скользящую дисперсию |

В следующем примере кода мы применим методы `Expr.ewm_mean()`, `Expr.rolling_mean()` и `Expr.rolling_min()` к столбцу `close` в датафрейме, содержащем информацию об акциях. Другие методы из этой таблицы работают аналогично:

```
stock = (
    pl.read_csv("data/stock/nvda/2023.csv", try_parse_dates=True)
    .select("date", "close")
    .with_columns(
        ewm_mean=pl.col("close").ewm_mean(com=7, ignore_nulls=True), ❶
        rolling_mean=pl.col("close").rolling_mean(window_size=7),
        rolling_min=pl.col("close").rolling_min(window_size=7),
    )
)

stock
```

❶ Аргумент `ignore_nulls=True` используется для более плавной обработки пропущенных значений в данных, что обеспечивает вычисление скользящего среднего на основе только фактических значений, без учета пропусков в данных.

Вывод:

```
shape: (124, 5)
```

| date | close | ewm_mean | rolling_mean | rolling_min |
|---|---|---|---|---|
| date | f64 | f64 | f64 | f64 |
| 2023-01-03 | 143.149994 | 143.149994 | null | null |
| 2023-01-04 | 147.490005 | 145.464667 | null | null |
| 2023-01-05 | 142.649994 | 144.398755 | null | null |
| 2023-01-06 | 148.589996 | 145.664782 | null | null |
| 2023-01-09 | 156.279999 | 148.388917 | null | null |
| … | … | … | … | … |
| 2023-06-26 | 406.320007 | 407.54911 | 425.805716 | 406.320007 |
| 2023-06-27 | 418.76001 | 408.950473 | 424.695718 | 406.320007 |
| 2023-06-28 | 411.170013 | 409.227915 | 422.445718 | 406.320007 |
| 2023-06-29 | 408.220001 | 409.101926 | 418.180006 | 406.320007 |
| 2023-06-30 | 423.019989 | 410.841684 | 417.118574 | 406.320007 |

Поскольку в таблице разницу между полученными значениями разглядеть непросто, давайте визуализируем результаты при помощи пакета plotnine (см. рис. 8.2):

```
from plotnine import *

(
    ggplot(stock.unpivot(index="date"), aes("date", "value", color="variable"))
    + geom_line(size=1)
    + labs(x="Дата", y="Значение", color="Метод")
    + theme_tufte(base_family="Guardian Sans", base_size=14)
    + theme(figure_size=(8, 5), dpi=200)
)
```

В главе 16 мы подробнее поговорим о создании визуализаций в библиотеке Polars.

## Операции сортировки

В нужном вам столбце данные отсортированы неправильно? Или, наоборот, они упорядочены, а должны быть перемешаны? Здесь вам на помощь придут перечисленные в табл. 8.10 методы.


**Рис. 8.2** ❖ Несколько скользящих операций, примененных к данным о ценах на акции

***Таблица 8.10. Методы сортировки***

| Метод | Описание |
|---|---|
| *Expr.arg_sort()* | Возвращает индексы, по которым можно отсортировать столбец |
| *Expr.shuffle()* | Выполняет перемешивание содержимого выражения |
| *Expr.sort()* | Сортирует столбец |
| *Expr.sort_by()* | Сортирует столбец по порядку, заданному в других столбцах |
| *Expr.rank()* | Ранжирует данные в столбце |
| *Expr.reverse()* | Обращает порядок следования элементов в столбце |

**Сортировка одного выражения – нечастое явление**

В реальных наборах данных строки зачастую представляют собой наблюдения или события. Так что обычно вам будет требоваться упорядочивать данные по целым строкам, чтобы атрибуты наблюдений не терялись. Однако в этом разделе мы говорим именно о сортировке одного выражения, или столбца. Методы сортировки датафреймов мы будем рассматривать в главе 11.

Давайте применим перечисленные методы к числовому ряду. Также мы добавили столбец y для демонстрации работы метода `Expr.sort_by()`:

```
(
    pl.DataFrame(
        {
            "x": [1, 3, None, 3, 7],
            "y": ["D", "I", "S", "C", "O"],
        }
```

```
    ).with_columns(
        arg_sort=pl.col("x").arg_sort(),
        shuffle=pl.col("x").shuffle(seed=7),
        sort=pl.col("x").sort(nulls_last=True),
        sort_by=pl.col("x").sort_by("y"),
        reverse=pl.col("x").reverse(),
        rank=pl.col("x").rank(),
    )
)
```

Вывод:

```
shape: (5, 8)
┌──────┬─────┬──────────┬─────────┬──────┬─────────┬─────────┬──────┐
│ x    ┆ y   ┆ arg_sort ┆ shuffle ┆ sort ┆ sort_by ┆ reverse ┆ rank │
│ ---  ┆ --- ┆ ---      ┆ ---     ┆ ---  ┆ ---     ┆ ---     ┆ ---  │
│ i64  ┆ str ┆ u32      ┆ i64     ┆ i64  ┆ i64     ┆ i64     ┆ f64  │
╞══════╪═════╪══════════╪═════════╪══════╪═════════╪═════════╪══════╡
│ 1    ┆ D   ┆ 2        ┆ 1       ┆ 1    ┆ 3       ┆ 7       ┆ 1.0  │
│ 3    ┆ I   ┆ 0        ┆ null    ┆ 3    ┆ 1       ┆ 3       ┆ 2.5  │
│ null ┆ S   ┆ 1        ┆ 3       ┆ 3    ┆ 3       ┆ null    ┆ null │
│ 3    ┆ C   ┆ 3        ┆ 7       ┆ 7    ┆ 7       ┆ 3       ┆ 2.5  │
│ 7    ┆ O   ┆ 4        ┆ 3       ┆ null ┆ null    ┆ 1       ┆ 4.0  │
└──────┴─────┴──────────┴─────────┴──────┴─────────┴─────────┴──────┘
```

# Прочие операции

В табл. 8.11 показан применяющийся к объектам Series метод, не вошедший ни в одну из указанных категорий.

***Таблица 8.11. Методы сортировки***

| Метод | Описание |
|---|---|
| *Expr.rle_id()* | Возвращает массив чисел, в котором каждое новое число соответствует изменению значения в исходном ряде. Таким образом мы получаем информацию о длинах серий в столбце |

Давайте применим метод `Expr.rle_id()` к числовому объекту Series:

```
(
    pl.DataFrame({"x": [33, 33, 27, 33, 60, 60, 60, 33, 60]}).with_columns(
        rle_id=pl.col("x").rle_id(),
    )
)
```

Вывод:

```
shape: (9, 2)
┌─────┬────────┐
│ x   ┆ rle_id │
│ --- ┆ ---    │
│ i64 ┆ u32    │
```

```
└─────┴────────┘
┌─────┬────────┐
│ 33  ┆ 0      │
│ 33  ┆ 0      │
│ 27  ┆ 1      │
│ 33  ┆ 2      │
│ 60  ┆ 3      │
│ 60  ┆ 3      │
│ 60  ┆ 3      │
│ 33  ┆ 4      │
│ 60  ┆ 5      │
└─────┴────────┘
```

Теперь поговорим о методах, применяющихся к объектам Series операции и снижающим их размерность до одной строки.

# Применяющиеся к объектам Series операции, снижающие размерность до одной строки

Продолжим рассматривать операции, которые применяются не поэлементно, а целиком ко всему объекту Series, но на этот раз поговорим о методах, агрегирующих все значения в объекте Series в одно. Примерами таких методов являются метод `Expr.mean()` для расчета среднего значения в столбце и метод `Expr.null_count()` для подсчета пропущенных значений.

В следующих четырех подразделах мы посмотрим на методы агрегирования, использующие кванторы, статистику, элементарный подсчет и другие техники.

### Повторяющиеся значения

При использовании операций агрегирования с сохранением каких-то из исходных столбцов рассчитанные значения будут повторяться в созданном столбце количество раз, соответствующее длине датафрейма. В примере ниже показано, как среднее значение по столбцу было повторено четыре раза в новом столбце:

```
(
    pl.DataFrame({"x": [1, 3, 3, 7]}).with_columns(
        mean=pl.col("x").mean(),
    )
)
```

Вывод:

```
shape: (4, 2)
┌─────┬──────┐
│ x   ┆ mean │
│ --- ┆ ---  │
│ i64 ┆ f64  │
```

```
└─────┴─────┘
┌─────┬─────┐
│ 1   ┆ 3.5 │
│ 3   ┆ 3.5 │
│ 3   ┆ 3.5 │
│ 7   ┆ 3.5 │
└─────┴─────┘
```

Поскольку операции агрегирования чаще всего применяются в контексте объединения данных, это обычно не является проблемой. Посмотрим, к примеру, что происходит при расчете среднего по группам:

```
(
    pl.DataFrame({"cluster": ["a", "a", "b", "b"], "x": [1, 3, 3, 7]})
    .group_by("cluster")
    .agg(
        mean=pl.col("x").mean(),
    )
)
```

Вывод:

```
shape: (2, 2)
┌─────────┬──────┐
│ cluster ┆ mean │
│ ---     ┆ ---  │
│ str     ┆ f64  │
╞═════════╪══════╡
│ a       ┆ 2.0  │
│ b       ┆ 5.0  │
└─────────┴──────┘
```

В оставшейся части раздела мы будем использовать метод датафрейма `select()` для исключения исходных столбцов во избежание повторения рассчитанных значений.

# Операции, использующие кванторы

Использование *кванторов* (quantifier) позволяет объединить несколько булевых значений в одно. В Polars реализованы *квантор общности* (universal quantifier) и *квантор существования* (existential quantifier) посредством методов `Expr.all()` и `Expr.any()`, представленных в табл. 8.12.

***Таблица 8.12. Методы, использующие кванторы***

| Метод | Описание |
|---|---|
| *`Expr.all()`* | Возвращает индикатор того, что все значения в столбце равны True |
| *`Expr.any()`* | Возвращает индикатор того, что хотя бы одно значение в столбце равно True |

Оба метода принимают аргумент `ignore_nulls`, говорящий о том, нужно ли игнорировать пропущенные значения. В следующем фрагменте кода мы применили методы `Expr.all()` и `Expr.any()` к трем булевым столбцам `x`, `y` и `z`:

```
df = pl.DataFrame(
    {
        "x": [True, False, False],
        "y": [True, True, True],
        "z": [False, False, False],
    }
)
print(df)
print(
    df.select(
        pl.all().all().name.suffix("_all"),
        pl.all().any().name.suffix("_any"),
    ),
)
```

Вывод:

```
shape: (3, 3)
┌───────┬──────┬───────┐
│ x     ┆ y    ┆ z     │
│ ---   ┆ ---  ┆ ---   │
│ bool  ┆ bool ┆ bool  │
╞═══════╪══════╪═══════╡
│ true  ┆ true ┆ false │
│ false ┆ true ┆ false │
│ false ┆ true ┆ false │
└───────┴──────┴───────┘
shape: (1, 6)
┌─────────┬─────────┬─────────┬─────────┬─────────┬─────────┐
│ x_all   ┆ y_all   ┆ z_all   ┆ x_any   ┆ y_any   ┆ z_any   │
│ ---     ┆ ---     ┆ ---     ┆ ---     ┆ ---     ┆ ---     │
│ bool    ┆ bool    ┆ bool    ┆ bool    ┆ bool    ┆ bool    │
╞═════════╪═════════╪═════════╪═════════╪═════════╪═════════╡
│ false   ┆ true    ┆ false   ┆ true    ┆ true    ┆ false   │
└─────────┴─────────┴─────────┴─────────┴─────────┴─────────┘
```

# Операции, вычисляющие статистику

В Polars реализовано множество методов для расчета различных статистических показателей. Все они перечислены в табл. 8.13.

***Таблица 8.13. Методы, вычисляющие статистику***

| Метод | Описание |
|---|---|
| *Expr.entropy()* | Вычисляет энтропию числового ряда |
| *Expr.kurtosis()* | Вычисляет коэффициент эксцесса, или куртозис (Фишера или Пирсона), для набора данных |
| *Expr.max()* | Вычисляет максимум |
| *Expr.mean()* | Вычисляет среднее значение |
| *Expr.median()* | Вычисляет медиану с помощью линейной интерполяции |

***Таблица 8.13*** *(окончание)*

| Метод | Описание |
|---|---|
| *Expr.min()* | Вычисляет минимум |
| *Expr.nan_max()* | Вычисляет максимум с включением встречающихся значений NaN |
| *Expr.nan_min()* | Вычисляет минимум с включением встречающихся значений NaN |
| *Expr.product()* | Вычисляет произведение выражения |
| *Expr.quantile()* | Вычисляет квантиль выражения |
| *Expr.skew()* | Вычисляет перекос (асимметрию) распределения |
| *Expr.std()* | Вычисляет стандартное отклонение |
| *Expr.sum()* | Вычисляет сумму |
| *Expr.var()* | Вычисляет дисперсию |

В примере ниже показано использование методов `Expr.max()`, `Expr.mean()`, `Expr.quantile()`, `Expr.skew()`, `Expr.std()`, `Expr.sum()` и `Expr.var()` применительно к числовому ряду из миллиона значений, принадлежащему нормальному распределению со средним значением 5 и стандартным отклонением 3:

```
samples = rng.normal(loc=5, scale=3, size=1_000_000)

(
    pl.DataFrame({"x": samples}).select(
        max=pl.col("x").max(),
        mean=pl.col("x").mean(),
        quantile=pl.col("x").quantile(quantile=0.95),
        skew=pl.col("x").skew(),
        std=pl.col("x").std(),
        sum=pl.col("x").sum(),
        var=pl.col("x").var(),
    )
)
```

Вывод:

```
shape: (1, 7)
┌────────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
│ max        ┆ mean     ┆ quantile ┆ skew     ┆ std      ┆ sum      ┆ var      │
│ ---        ┆ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      │
│ f64        ┆ f64      ┆ f64      ┆ f64      ┆ f64      ┆ f64      ┆ f64      │
╞════════════╪══════════╪══════════╪══════════╪══════════╪══════════╪══════════╡
│ 20.752443  ┆ 4.994978 ┆ 9.931565 ┆ 0.003245 ┆ 2.999926 ┆ 4.9950e6 ┆ 8.999558 │
└────────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────┘
```

Методы `Expr.entropy()`, `Expr.kurtosis()`, `Expr.median()`, `Expr.min()`, `Expr.nan_max()`, `Expr.nan_min()` и `Expr.product()` применяются аналогично.

# Операции подсчета

В Polars реализовано несколько методов подсчета различных значений, включая уникальные и пропущенные. Все они перечислены в табл. 8.14.

***Таблица 8.14. Методы подсчета***

| Метод | Описание |
|---|---|
| *Expr.approx_n_unique()* | Рассчитывает приблизительное количество уникальных значений |
| *Expr.count()* | Рассчитывает количество элементов в выражении без учета пропущенных значений |
| *Expr.len()* | Рассчитывает количество элементов в выражении с учетом пропущенных значений |
| *Expr.n_unique()* | Рассчитывает точное количество уникальных значений |
| *Expr.null_count()* | Рассчитывает количество пропущенных значений |

Для демонстрации этих методов сгенерируем числовой ряд из случайных целочисленных значений в диапазоне от 0 до 1000 в количестве 1729 штук с одним пропущенным значением:

```
samples = pl.Series(rng.integers(low=0, high=10_000, size=1_729))
samples[403] = None ❶
df_ints = pl.DataFrame({"x": samples}).with_row_index() ❷
df_ints.slice(400, 6) ❸
```

❶ Элемент с индексом 403 сделаем пропущенным.
❷ С помощью метода *with_row_index()* можно добавить в датафрейм столбец с порядковыми номерами строк.
❸ Воспользуемся методом *slice()* для вывода подмножества строк в датафрейме.

Вывод:

```
shape: (6, 2)
┌───────┬──────┐
│ index ┆ x    │
│ ---   ┆ ---  │
│ u32   ┆ i64  │
╞═══════╪══════╡
│ 400   ┆ 807  │
│ 401   ┆ 8634 │
│ 402   ┆ 2109 │
│ 403   ┆ null │
│ 404   ┆ 1740 │
│ 405   ┆ 3333 │
└───────┴──────┘
```

Теперь применим упомянутые пять методов к столбцу x:

```
df_ints.select(
    approx_n_unique=pl.col("x").approx_n_unique(),
    count=pl.col("x").count(),
    len=pl.col("x").len(),
    n_unique=pl.col("x").n_unique(),
    null_count=pl.col("x").null_count(),
)
```

Вывод:

```
shape: (1, 5)
┌──────────────────┬─────────┬─────────┬──────────┬────────────┐
│ approx_n_unique  ┆ count   ┆ len     ┆ n_unique ┆ null_count │
│ ---              ┆ ---     ┆ ---     ┆ ---      ┆ ---        │
│ u32              ┆ u32     ┆ u32     ┆ u32      ┆ u32        │
╞══════════════════╪═════════╪═════════╪══════════╪════════════╡
│ 1572             ┆ 1728    ┆ 1729    ┆ 1575     ┆ 1          │
└──────────────────┴─────────┴─────────┴──────────┴────────────┘
```

Вас может удивить целесообразность присутствия метода приблизительного подсчета уникальных значений `Expr.approx_n_unique()` при наличии метода точного подсчета `Expr.n_unique()`.

На самом деле этот метод выполняется гораздо быстрее в сравнении с `Expr.n_unique()`, так что в ситуациях, когда вам важна не точность подсчета, а скорость, он может стать незаменимой альтернативой. Давайте посмотрим на разницу в скорости на примере объемного датафрейма:

```
large_df_ints = pl.DataFrame(
    {"x": rng.integers(low=0, high=10_000, size=10_000_000)}
)
```

Замерим время выполнения метода `Expr.n_unique()`:

```
%%time
large_df_ints.select(pl.col("x").n_unique())
```

Вывод:

```
CPU times: user 155 ms, sys: 21.6 ms, total: 177 ms
Wall time: 54.5 ms
shape: (1, 1)
┌─────────┐
│ x       │
│ ---     │
│ u32     │
╞═════════╡
│ 10000   │
└─────────┘
```

А теперь применим метод `Expr.approx_n_unique()`:

```
%%time
large_df_ints.select(pl.col("x").approx_n_unique())
```

Вывод:

```
CPU times: user 26.8 ms, sys: 50 µs, total: 26.8 ms
Wall time: 26.8 ms
shape: (1, 1)
┌─────────┐
│ x       │
│ ---     │
```

```
| u32   |
╞═══════╡
| 10013 |
└───────┘
```

В точности вычислений мы недосчитались всего порядка 0.1 %, но при этом смогли добиться существенно большего быстродействия.

## Прочие операции

Есть еще восемь операций, применяющихся к объектам Series и сводящих результат в одну строку, которые не попали ни в одну из приведенных категорий. Методы для реализации этих операций приведены в табл. 8.15.

***Таблица 8.15. Прочие методы, сводящие результат в одну строку***

| Метод | Описание |
|---|---|
| *Expr.arg_max()* | Возвращает индекс максимального значения |
| *Expr.arg_min()* | Возвращает индекс минимального значения |
| *Expr.first()* | Возвращает первое значение |
| *Expr.get()* | Возвращает значение по индексу |
| *Expr.implode()* | Агрегирует значения в список |
| *Expr.last()* | Возвращает последнее значение |
| *Expr.lower_bound()* | Рассчитывает нижнюю границу |
| *Expr.upper_bound()* | Рассчитывает верхнюю границу |

Применим методы `Expr.arg_min()`, `Expr.first()`, `Expr.get()`, `Expr.implode()`, `Expr.last()` и `Expr.upper_bound()` к набору данных из предыдущего примера. Метод `Expr.arg_max()` применяется аналогично методу `Expr.arg_min()`, а `Expr.lower_bound()` – аналогично методу `Expr.upper_bound()`:

```
df_ints.select(
    arg_min=pl.col("x").arg_min(),
    first=pl.col("x").first(),
    get=pl.col("x").get(403), ❶
    implode=pl.col("x").implode(),
    last=pl.col("x").last(),
    upper_bound=pl.col("x").upper_bound(),
)
```

❶ В результате мы получили значение `null`, поскольку сделали значение с индексом 403 пропущенным.

Вывод:

```
shape: (1, 6)
┌─────────┬───────┬──────┬─────────────┬──────┬─────────────┐
| arg_min | first | get  | implode     | last | upper_bound |
| ---     | ---   | ---  | ---         | ---  | ---         |
```

```
| u32     | i64   | i64  | list[i64]          | i64  | i64                  |
╞═════════╪═══════╪══════╪════════════════════╪══════╪══════════════════════╡
| 0       | 0     | null | [0, 7245, … 3723] | 3723 | 9223372036854775807 |
└─────────┴───────┴──────┴────────────────────┴──────┴──────────────────────┘
```

# Применяющиеся к объектам Series операции, снижающие размерность до одной или нескольких строк

Помимо методов, применяющихся к объектам Series и снижающих их размерность до одной строки, в Polars присутствуют и методы, снижающие размерность исходного объекта до одной или нескольких строк. В следующих четырех разделах мы рассмотрим такие методы.

## Операции, связанные с уникальными значениями

В табл. 8.16 представлены методы для работы с уникальными значениями в наборе данных.

***Таблица 8.16. Методы для работы с уникальными значениями***

| Метод | Описание |
|---|---|
| *Expr.arg_unique()* | Возвращает индекс первого уникального значения |
| *Expr.unique()* | Возвращает уникальные значения в выражении |
| *Expr.unique_counts()* | Возвращает количество уникальных значений в порядке их появления |
| *Expr.value_counts()* | Подсчитывает вхождения уникальных значений |

Применим эти методы к строковому объекту Series:

```
(
    pl.DataFrame({"x": ["A", "C", "D", "C"]}).select(
        arg_unique=pl.col("x").arg_unique(),
        unique=pl.col("x").unique(maintain_order=True), ❶
        unique_counts=pl.col("x").unique_counts(),
        value_counts=pl.col("x").value_counts(sort=True), ❷
    )
)
```

❶ Сохранение порядка следования значений может быть более затратным в плане вычислений.

❷ Метод `Expr.value_counts()` возвращает результат в виде значения с типом данных `pl.Struct`, представляющим собой комбинацию результатов методов `Expr.unique()` и `Expr.unique_counts()`, хоть и не всегда в том же порядке[1].

Вывод:

```
shape: (3, 4)
┌─────────────┬────────┬─────────────────┬──────────────┐
│ arg_unique  ┆ unique ┆ unique_counts   ┆ value_counts │
│ ---         ┆ ---    ┆ ---             ┆ ---          │
│ u32         ┆ str    ┆ u32             ┆ struct[2]    │
╞═════════════╪════════╪═════════════════╪══════════════╡
│ 0           ┆ A      ┆ 1               ┆ {"C",2}      │
│ 1           ┆ C      ┆ 2               ┆ {"A",1}      │
│ 2           ┆ D      ┆ 1               ┆ {"D",1}      │
└─────────────┴────────┴─────────────────┴──────────────┘
```

# Операции отбора

Иногда нам необходимо просто извлечь данные из набора в том или ином виде. В табл. 8.17 приведены методы для отбора элементов на основе их позиций или значений.

***Таблица 8.17. Методы отбора***

| Метод | Описание |
|---|---|
| *Expr.bottom_k()* | Возвращает k наименьших значений |
| *Expr.head()* | Возвращает первые n строк |
| *Expr.limit()* | Возвращает первые n строк (алиас для метода `Expr.head()`) |
| *Expr.sample()* | Возвращает выборку из выражения |
| *Expr.slice()* | Возвращает срез из выражения |
| *Expr.tail()* | Возвращает последние n строк |
| *Expr.gather()* | Возвращает значения по индексам |
| *Expr.gather_every()* | Извлекает каждый n-й элемент в объекте Series и возвращает результат в виде нового объекта Series |
| *Expr.top_k* | Возвращает k наибольших значений |

Ниже показан пример применения методов `Expr.bottom_k()`, `Expr.head()`, `Expr.sample()`, `Expr.slice()`, `Expr.gather()`, `Expr.gather_every()` и `Expr.top_k()` с нашим предыдущим набором данных.

---

[1] Поскольку в библиотеке Polars повсюду применяются параллельные вычисления, результаты могут оказаться не детерминированы заранее, все зависит от времени окончания операций.

Метод `Expr.limit()` является алиасом для метода `Expr.head()`. Метод `Expr.tail()` работает аналогично методу `Expr.head()`, но возвращает не первые строки в наборе, а последние:

```
df_ints.select(
    bottom_k=pl.col("x").bottom_k(7),
    head=pl.col("x").head(7),
    sample=pl.col("x").sample(7),
    slice=pl.col("x").slice(400, 7),
    gather=pl.col("x").gather([1, 1, 2, 3, 5, 8, 13]),
    gather_every=pl.col("x").gather_every(247), ❶
    top_k=pl.col("x").top_k(7),
)
```

❶ Новый объект Series должен состоять из семи элементов, иначе будет возвращена ошибка несовпадения длин объектов. В нашем примере если взять каждый 247-й элемент из объекта Series длины 1729, мы как раз и получим семь значений.

Вывод:

```
shape: (7, 7)
┌──────────┬──────┬────────┬───────┬────────┬──────────────┬──────┐
│ bottom_k ┆ head ┆ sample ┆ slice ┆ gather ┆ gather_every ┆ top_k │
│ ---      ┆ ---  ┆ ---    ┆ ---   ┆ ---    ┆ ---          ┆ ---  │
│ i64      ┆ i64  ┆ i64    ┆ i64   ┆ i64    ┆ i64          ┆ i64  │
╞══════════╪══════╪════════╪═══════╪════════╪══════════════╪══════╡
│ 0        ┆ 0    ┆ 6871   ┆ 807   ┆ 7245   ┆ 0            ┆ 9998 │
│ 1        ┆ 7245 ┆ 2202   ┆ 8634  ┆ 7245   ┆ 8680         ┆ 9988 │
│ 6        ┆ 5227 ┆ 7328   ┆ 2109  ┆ 5227   ┆ 8483         ┆ 9988 │
│ 7        ┆ 2747 ┆ 1648   ┆ null  ┆ 2747   ┆ 8358         ┆ 9986 │
│ 10       ┆ 9816 ┆ 5761   ┆ 1740  ┆ 2657   ┆ 1805         ┆ 9985 │
│ 21       ┆ 2657 ┆ 9315   ┆ 3333  ┆ 5393   ┆ 3638         ┆ 9979 │
│ 29       ┆ 4578 ┆ 8370   ┆ 788   ┆ 8203   ┆ 5843         ┆ 9975 │
└──────────┴──────┴────────┴───────┴────────┴──────────────┴──────┘
```

## Операции по удалению пропущенных значений

Иногда нам необходимо избавиться в наборе данных от пустых значений. В табл. 8.18 приведены методы для борьбы с нечисловыми и пропущенными значениями.

***Таблица 8.18. Методы для удаления нечисловых и пропущенных значений***

| Метод | Описание |
|---|---|
| *`Expr.drop_nans()`* | Удаляет значения `NaN` |
| *`Expr.drop_nulls()`* | Удаляет значения `null` |

Вот как вы можете применить эти методы на практике:

```
x = [None, 1.0, 2.0, 3.0, np.nan]
(
```

```
    pl.DataFrame({"x": x}).select(
        drop_nans=pl.col("x").drop_nans(), drop_nulls=pl.col("x").drop_nulls()
    )
)
```

Вывод:

```
shape: (4, 2)
┌────────────┬─────────────┐
│ drop_nans  ┆ drop_nulls  │
│ ---        ┆ ---         │
│ f64        ┆ f64         │
╞════════════╪═════════════╡
│ null       ┆ 1.0         │
│ 1.0        ┆ 2.0         │
│ 2.0        ┆ 3.0         │
│ 3.0        ┆ NaN         │
└────────────┴─────────────┘
```

# Прочие операции

Существует еще шесть методов, сводящих исходный набор данных к одной или нескольким строкам. Эти методы показаны в табл. 8.19.

***Таблица 8.19. Прочие методы, снижающие размерность до одной или нескольких строк***

| Метод | Описание |
|---|---|
| *Expr.arg_true()* | Возвращает индексы элементов, в которых выражение дает значение `True` |
| *Expr.flatten()* | Выравнивает список или столбец, состоящий из строк |
| *Expr.mode()* | Вычисляет наиболее часто встречающиеся значения |
| *Expr.reshape()* | Приводит форму выражения к плоскому объекту Series или объекту Series, содержащему списки |
| *Expr.rle()* | Возвращает длины серий одинаковых значений |
| *Expr.search_sorted()* | Обнаруживает индексы, в которые необходимо вставить элементы для поддержания порядка |

Давайте применим методы `Expr.arg_true()`, `Expr.mode()`, `Expr.reshape()`, `Expr.rle()` и `Expr.search_sorted()` к неупорядоченному объекту Series, состоящему из целочисленных значений. Мы продемонстрируем применение методов по отдельности, поскольку они генерируют объекты Series разной длины.

Сначала покажем на практике метод `Expr.arg_true()`:

```
numbers = [33, 33, 27, 33, 60, 60, 60, 33, 60]

(
    pl.DataFrame({"x": numbers}).select(
        arg_true=(pl.col("x") >= 60).arg_true(), ❶
    )
)
```

❶ Мы воспользовались оператором больше или равно (>=) для получения булева объекта Series. В главе 9 мы подробно поговорим об этом и других операторах сравнения.

Вывод:

```
shape: (4, 1)
┌──────────┐
│ arg_true │
│ ---      │
│ u32      │
╞══════════╡
│ 4        │
│ 5        │
│ 6        │
│ 8        │
└──────────┘
```

Теперь применим метод `Expr.mode()`:

```
(
    pl.DataFrame({"x": numbers}).select(
        mode=pl.col("x").mode().sort(),
    )
)
```

Вывод:

```
shape: (2, 1)
┌──────┐
│ mode │
│ ---  │
│ i64  │
╞══════╡
│ 33   │
│ 60   │
└──────┘
```

Третьим воспользуемся методом `Expr.reshape()`, как показано ниже:

```
(
    pl.DataFrame({"x": numbers}).select(
        reshape=pl.col("x").reshape((3, 3)), ❶
    )
)
```

❶ Общее количество элементов должно остаться неизменным. К примеру, мы бы не смогли свести форму таких данных к пяти строкам, в последней из которых располагался бы объект `pl.List`, состоящий из одного элемента.

Вывод:

```
shape: (3, 1)
┌──────────────┐
│ reshape      │
│ ---          │
│ array[i64, 3] │
╞══════════════╡
│ [33, 33, 27] │
│ [33, 60, 60] │
│ [60, 33, 60] │
└──────────────┘
```

Теперь применим метод `Expr.rle()` для нахождения длин серий одинаковых значений:

```
(
    pl.DataFrame({"x": numbers}).select(
        rle=pl.col("x").rle(), ❶
    )
)
```

❶ Сравните этот метод с методом `Expr.rle_id()`, который мы обсуждали ранее.

Вывод:

```
shape: (6, 1)
┌──────────┐
│ rle      │
│ ---      │
│ struct[2] │
╞══════════╡
│ {2,33}   │
│ {1,27}   │
│ {1,33}   │
│ {3,60}   │
│ {1,33}   │
│ {1,60}   │
└──────────┘
```

Наконец, воспользуемся методом `Expr.search_sorted()`:

```
(
    pl.DataFrame({"x": numbers}).select(
        rle=pl.col("x").sort().search_sorted(42), ❶
    )
)
```

❶ Метод `Expr.search_sorted()`, вероятно, является наиболее полезным применительно к упорядоченным объектам Series.

Вывод:

```
shape: (1, 1)
┌─────┐
│ rle │
│ --- │
│ u32 │
╞═════╡
│ 5   │
└─────┘
```

Больше о применении метода `Expr.flatten()` вы узнаете в главе 12, поскольку он является алиасом для метода `Expr.list.explode()`.

# Применяющиеся к объектам Series операции, увеличивающие размерность

В Polars реализованы две операции, приводящие к увеличению размерности исходного объекта Series. Они показаны в табл. 8.20.

***Таблица 8.20. Методы, приводящие к увеличению размерности объекта Series***

| Метод | Описание |
|---|---|
| *`Expr.explode()`* | Раскрывает списки в выражении |
| *`Expr.extend_constant()`* | Расширяет объект Series за счет новых значений |

Давайте воспользуемся методом `Expr.explode()`, чтобы превратить колонку со списками в виде значений в столбец с отдельными значениями из этих списков:

```
(
    pl.DataFrame(
        {
            "x": [["a", "b"], ["c", "d"]],
        }
    ).select(explode=pl.col("x").explode())
)
```

Вывод:

```
shape: (4, 1)
┌─────────┐
│ explode │
│ ---     │
│ str     │
╞═════════╡
│ a       │
```

```
| b       |
| c       |
| d       |
└─────────┘
```

Пример с применением метода `Expr.extend_constant()` мы видели в начале этой главы.

# Заключение

В этой главе мы познакомились со множеством реализованных в библиотеке Polars выражений. Вы узнали, что:

- выражения в Polars могут расширяться за счет применения операций, таких как математическое преобразование, обработка пропущенных значений и применение статистических вычислений;
- операции бывают поэлементными и обрабатывающими объекты Series целиком. Также они могут приводить к уменьшению, сохранению и увеличению размерности исходного объекта Series;
- поэлементные операции применяются к каждому значению независимо и включают такие методы, как, например, `Expr.sqrt()`, `Expr.log()` и `Expr.sin()`. Примерами операций по работе с объектами Series являются методы `Expr.cum_sum()` и `Expr.rolling_mean()`;
- применение методов вроде `Expr.unique()`, `Expr.mean()` или `Expr.is_null()` приводит к агрегации значений в исходном объекте Series и получению в результате одного или нескольких значений;
- с пропущенными значениями в наборе данных можно бороться в Polars при помощи методов `Expr.fill_nan()` и `Expr.fill_null()`, тогда как для работы с дубликатами существуют методы, такие как `Expr.is_duplicated()` и `Expr.is_unique()`;
- расчет скользящих показателей при помощи таких методов, как `Expr.rolling_mean()` и `Expr.rolling_sum()`, может быть полезен при необходимости сгладить паразитные колебания в данных;
- упорядочивание значений в нужном вам порядке можно реализовать при помощи методов, подобных `Expr.sort()`, `Expr.reverse()` и `Expr.rank()`;
- операции, расширяющие исходный объект Series, реализованы в Polars в виде двух методов: `Expr.extend_constant()` и `Expr.explode()`.

В следующей главе мы научимся комбинировать выражения.

# Глава 9

# Комбинирование выражений

Теперь, когда вы понимаете основы работы с выражениями в Polars и знаете их основные методы, пришло время научиться объединять выражения.

Комбинирование выражений может понадобиться в случаях, когда создаваемый объект Series базируется более чем на одном значении или столбце. А это может происходить гораздо чаще, чем вам кажется. Например, когда вам необходимо вычислить соотношение между двумя вещественными столбцами, отфильтровать строки по нескольким условиям или объединить несколько строковых столбцов в один.

На самом деле вы уже встречались с комбинированием выражений в предыдущих главах. Взгляните на пример из главы 7:

```
fruit = pl.read_csv("data/fruit.csv")
fruit.filter(pl.col("is_round") & (pl.col("weight") > 1000))
```

Вывод:

```
shape: (2, 5)
┌────────────┬────────┬────────┬──────────┬────────┐
│ name       ┆ weight ┆ color  ┆ is_round ┆ origin │
│ ---        ┆ ---    ┆ ---    ┆ ---      ┆ ---    │
│ str        ┆ i64    ┆ str    ┆ bool     ┆ str    │
╞════════════╪════════╪════════╪══════════╪════════╡
│ Cantaloupe ┆ 2500   ┆ orange ┆ true     ┆ Africa │
│ Watermelon ┆ 5000   ┆ green  ┆ true     ┆ Africa │
└────────────┴────────┴────────┴──────────┴────────┘
```

Здесь мы объединяем два существующих столбца (`is_round` и `weight`) и одно значение (`1000`) в одно выражение. После этого метод датафрейма `filter()` использует это выражение для отбора нужных строк.

По тому, как расставлены скобки, понятно, что действия выполняются в следующем порядке. Сначала столбец `weight` и значение `1000` объединяются при помощи оператора сравнения *больше* (`>`). Если условие выполняется,

возвращается значение `True`. Далее с помощью логического оператора И (`&`) столбец `is_round` объединяется с объектом Series, полученным на первом шаге. И только в случае, если обе составляющие истинны, мы получим результирующее значение `True`, которое метод `filter()` воспринимает как призыв к тому, чтобы оставить строку в итоговом наборе.

**Приоритет операторов**

Выполнение операторов в Polars осуществляется в порядке, принятом в Python. Таблицу с приоритетами операторов можно найти по адресу *https://docs.python.org/3/reference/expressions.html#operator-precedence*.

Но это лишь вершина айсберга, когда речь идет о комбинировании выражений в Polars. В этой главе вы узнаете:

- какие различия существуют между встраиваемыми операторами и цепочками методов;
- как комбинировать выражения:
  - путем применения арифметических операций, таких как сложение и умножение;
  - с помощью операций сравнения;
  - посредством булевой алгебры;
  - с применением битовых операций, таких как AND и XOR;
  - с помощью разных функций уровня модуля.

Все инструкции по загрузке и установке требуемых файлов вы найдете в главе 2. Мы будем исходить из предположения о том, что все рабочие файлы вы поместили в директорию *data*.

# Встраиваемые операторы против методов

В предыдущих двух главах мы использовали способ объединения методов в цепочки для написания сложных выражений. Однако вместо цепочек методов для комбинирования выражений можно также использовать встраиваемые операторы. Оба способа дают один и тот же результат, что видно на рис. 9.1.

**Методы и операторы**

Хотя каждый оператор имеет соответствующий ему метод объекта `Expr`, не каждый метод (или функция) для комбинирования выражений может похвастаться наличием соответствующего оператора. В качестве примеров можно привести метод `Expr.dot()` и функцию `pl.concat_list()`.

Для иллюстрации давайте перемножим значения в двух столбцах с использованием двух подходов:

```
(
    pl.DataFrame({"i": [6.0, 0, 2, 2.5], "j": [7.0, 1, 2, 3]}).with_columns(
        (pl.col("i") * pl.col("j")).alias("*"),
        pl.col("i").mul(pl.col("j")).alias("Expr.mul()"),
    )
)
```

Вывод:

```
shape: (4, 4)
┌─────┬─────┬──────┬────────────┐
│ i   ┆ j   ┆ *    ┆ Expr.mul() │
│ --- ┆ --- ┆ ---  ┆ ---        │
│ f64 ┆ f64 ┆ f64  ┆ f64        │
╞═════╪═════╪══════╪════════════╡
│ 6.0 ┆ 7.0 ┆ 42.0 ┆ 42.0       │
│ 0.0 ┆ 1.0 ┆ 0.0  ┆ 0.0        │
│ 2.0 ┆ 2.0 ┆ 4.0  ┆ 4.0        │
│ 2.5 ┆ 3.0 ┆ 7.5  ┆ 7.5        │
└─────┴─────┴──────┴────────────┘
```

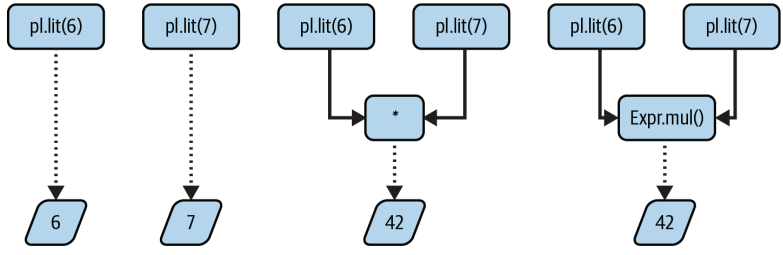



**Рис. 9.1** ❖ Комбинирование выражений при помощи встраиваемых операторов и цепочек методов дает один и тот же результат

Как и ожидалось, оба подхода дали один и тот же результат. Нам кажется, что пробелы с двух сторон от встраиваемого оператора (*) лучше подчеркивают тот факт, что вы объединяете два выражения в одно. По этой причине мы обычно рекомендуем использовать операторы, а не методы, если они существуют. Но обратите внимание, что при использовании операторов вам необходимо заключить объединенное выражение в скобки, чтобы применить к нему дополнительные методы, – в нашем случае это метод `Expr.alias()`.

Вот так просто можно перемножить два выражения. Теперь взглянем на другие арифметические операторы.

# Арифметические операторы

Арифметика является основой основ любого анализа данных. Вы можете выполнять арифметические операции применительно к выражениям и значениям в Python.

Приведем пример с нашим датафреймом с фруктами и разделим значения в столбце `weight` (выражение) на 1000 (значение в Python) при помощи *арифметического оператора* `/`:

```
fruit.select(pl.col("name"), (pl.col("weight") / 1000))
```

Вывод:

```
shape: (10, 2)
┌─────────────┬────────┐
│ name        ┆ weight │
│ ---         ┆ ---    │
│ str         ┆ f64    │
╞═════════════╪════════╡
│ Avocado     ┆ 0.2    │
│ Banana      ┆ 0.12   │
│ Blueberry   ┆ 0.001  │
│ Cantaloupe  ┆ 2.5    │
│ Cranberry   ┆ 0.002  │
│ Elderberry  ┆ 0.001  │
│ Orange      ┆ 0.13   │
│ Papaya      ┆ 1.0    │
│ Peach       ┆ 0.15   │
│ Watermelon  ┆ 5.0    │
└─────────────┴────────┘
```

В табл. 9.1 приведены встраиваемые операторы и соответствующие им методы для реализации арифметических операций в Polars.

***Таблица 9.1. Операторы и методы для реализации арифметических операций в Polars***

| Оператор | Метод | Описание |
|---|---|---|
| `+` | *`Expr.add()`* | Сложение |
| `-` | *`Expr.sub()`* | Вычитание |
| `*` | *`Expr.mul()`* | Умножение |
| `/` | *`Expr.truediv()`* | Деление |
| `//` | *`Expr.floordiv()`* | Целочисленное деление |
| `**` | *`Expr.pow()`* | Возведение в степень |
| `%` | *`Expr.mod()`* | Остаток от деления |
| отсутствует | *`Expr.dot()`* | Скалярное произведение |

В следующем фрагменте кода мы посмотрим, как можно применять эти встраиваемые операторы с целочисленными столбцами `i` и `j`. При необходимости Polars автоматически создает вещественный столбец. Поскольку у метода `Expr.dot()` отсутствует аналог в виде оператора, мы применили именно его:

```
pl.Config(float_precision=2, tbl_cell_numeric_alignment="RIGHT") ❶

(
    pl.DataFrame({"i": [0.0, 2, 2, -2, -2], "j": [1, 2, 3, 4, -5]}).with_columns(
```

```
        (pl.col("i") + pl.col("j")).alias("i + j"),
        (pl.col("i") - pl.col("j")).alias("i - j"),
        (pl.col("i") * pl.col("j")).alias("i * j"),
        (pl.col("i") / pl.col("j")).alias("i / j"),
        (pl.col("i") // pl.col("j")).alias("i // j"),
        (pl.col("i") ** pl.col("j")).alias("i ** j"),
        (pl.col("j") % 2).alias("j % 2"), ❷
        pl.col("i").dot(pl.col("j")).alias("i · j"), ❸
    )
)
```

❶ Мы временно изменили эти две опции вывода, чтобы результат поместился на страницу.

❷ Оператор нахождения остатка от деления может принимать второе выражение, как и остальные арифметические операторы.

❸ В отсутствие аналога для операции скалярного произведения в виде встраиваемого оператора мы воспользовались знаком точки (·) при именовании нового столбца. Скалярное произведение представляет собой математическую операцию, заключающуюся в поэлементном перемножении элементов двух векторов и суммировании результатов. В данном случае эта операция была применена ко всем столбцам целиком, а не к отдельным значениям. Именно поэтому в последнем столбце у нас оказались одинаковые значения (12.0).

Вывод:

```
shape: (5, 10)
```

| i | j | i + j | i - j | i * j | i / j | i // j | i ** j | j % 2 | i · j |
|---|---|---|---|---|---|---|---|---|---|
| f64 | i64 | f64 | f64 | f64 | f64 | f64 | f64 | i64 | f64 |
| 0.00 | 1 | 1.00 | -1.00 | 0.00 | 0.00 | 0.00 | 0.00 | 1 | 12.00 |
| 2.00 | 2 | 4.00 | 0.00 | 4.00 | 1.00 | 1.00 | 4.00 | 0 | 12.00 |
| 2.00 | 3 | 5.00 | -1.00 | 6.00 | 0.67 | 0.00 | 8.00 | 1 | 12.00 |
| -2.00 | 4 | 2.00 | -6.00 | -8.00 | -0.50 | -1.00 | 16.00 | 0 | 12.00 |
| -2.00 | -5 | -7.00 | 3.00 | 10.00 | 0.40 | 0.00 | -0.03 | 1 | 12.00 |

# Операторы сравнения

Какой из проведенных экспериментов дал значимый результат? Какие фильмы, выпущенные в 90-х, имеют рейтинг на сайте IMDB 8.7 и выше? Не превышает ли вольтаж допустимые значения? На все эти вопросы можно ответить при помощи *операторов сравнения*.

Операторы сравнения в Polars работают аналогично схожим операторам в Python, за исключением того, что их нельзя объединять в цепочки (об этом далее):

```
pl.select(pl.lit("a") > pl.lit("b"))
```

Вывод:

```
shape: (1, 1)
┌─────────┐
│ literal │
│ ---     │
│ bool    │
╞═════════╡
│ false   │
└─────────┘
```

Чаще всего вы будете сравнивать друг с другом целочисленные или вещественные значения, такие как `pl.Int8` или `pl.Float64`. Но вы также можете выполнять сравнение строковых и временных значений, описываемых типами данных `pl.Date`, `pl.Datetime`, `pl.Duration` и `pl.Time`.

Сравнение всегда приводит к образованию булева объекта Series. Этот столбец может быть добавлен к датафрейму, но чаще его используют в качестве маски для фильтрации датафрейма с помощью метода `filter()` или в условных выражениях с использованием функции `pl.when()`.

Ниже приведен пример с нашим датафреймом `fruit`, в котором мы сравниваем значения в столбце `weight` со значением 1000 при помощи оператора *больше или равно* (`>=`), а созданный в результате булев объект Series используем для фильтрации строк в наборе:

```
(
    fruit.select(
    pl.col("name"),
    pl.col("weight"),
    ).filter(pl.col("weight") >= 1000)
)
```

Вывод:

```
shape: (3, 2)
┌─────────────┬────────┐
│ name        ┆ weight │
│ ---         ┆ ---    │
│ str         ┆ i64    │
╞═════════════╪════════╡
│ Cantaloupe  ┆ 2500   │
│ Papaya      ┆ 1000   │
│ Watermelon  ┆ 5000   │
└─────────────┴────────┘
```

В табл. 9.2 перечислены операторы сравнения, реализованные в Polars.

*Таблица 9.2. Операторы и методы для реализации операций сравнения в Polars*

| Оператор | Метод | Описание |
|---|---|---|
| `<` | *`Expr.lt()`* | Меньше |
| `<=` | *`Expr.le()`* | Меньше или равно |
| `==` | *`Expr.eq()`* | Равно |
| `>=` | *`Expr.ge()`* | Больше или равно |
| `>` | *`Expr.gt()`* | Больше |
| `!=` | *`Expr.ne()`* | Не равно |

## Объединение операторов сравнения

В Python допустимо объединять встраиваемые операторы сравнения, приведенные в табл. 9.2. Рассмотрим следующий пример, в котором оператор `<` используется дважды для проверки того, лежит ли значение `x` в диапазоне от 3 до 5:

```
x = 4
3 < x < 5
```

Вывод:

```
True
```

Если сделать нечто подобное в Polars, вы получите ошибку:

```
pl.select(pl.lit(3) < pl.lit(x) < pl.lit(5))
```

Вывод:

```
TypeError: the truth value of an Expr is ambiguous

You probably got here by using a Python standard library function instead of
the native expressions API.
Here are some things you might want to try:
- instead of `pl.col('a') and pl.col('b')`, use `pl.col('a') & pl.col('b')`
- instead of `pl.col('a') in [y, z]`, use `pl.col('a').is_in([y, z])`
- instead of `max(pl.col('a'), pl.col('b'))`, use `pl.max_horizontal(pl.col(
'a'), pl.col('b'))`
```

Решение состоит в том, чтобы использовать два отдельных оператора сравнения и объединить результат с помощью оператора логического И (`&`), как показано ниже:

```
pl.select((pl.lit(3) < pl.lit(x)) & (pl.lit(x) < pl.lit(5))).item()
```

Вывод:

```
True
```

О логических операторах, подобных `&`, мы будем говорить в следующем разделе, посвященном булевой алгебре. Также вы можете поискать в API Polars подходящий метод, реализующий нужную вам логику. В данном случае вы можете воспользоваться методом *`Expr.is_between()`* следующим образом:

```
pl.select(pl.lit(x).is_between(3, 5)).item()
```

Вывод:

```
True
```

Давайте применим несколько операторов сравнения к двум числовым столбцам a и b:

```
(
    pl.DataFrame(
        {"a": [-273.15, 0, 42, 100], "b": [1.4142, 2.7183, 42, 3.1415]}
    ).with_columns(
        (pl.col("a") == pl.col("b")).alias("a == b"),
        (pl.col("a") <= pl.col("b")).alias("a <= b"),
        (pl.all() > 0).name.suffix(" > 0"),
        ((pl.col("b") - pl.lit(2).sqrt()).abs() < 1e-3).alias("b ≈ √2"), ❶
        ((1 < pl.col("b")) & (pl.col("b") < 3)).alias("1 < b < 3"),
    )
)
```

❶ Здесь мы воспользовались арифметическими операторами и операторами сравнения для комбинирования выражений.

Вывод:

```
shape: (4, 8)
```

| a | b | a == b | a <= b | a > 0 | b > 0 | b ≈ √2 | 1 < b < 3 |
|---|---|---|---|---|---|---|---|
| f64 | f64 | bool | bool | bool | bool | bool | bool |
| -273.15 | 1.4142 | false | true | false | true | true | true |
| 0.0 | 2.7183 | false | true | false | true | false | true |
| 42.0 | 42.0 | true | true | true | true | false | false |
| 100.0 | 3.1415 | false | false | true | true | false | false |

В следующем примере демонстрируется применение операторов сравнения к значениям разных типов. Два варианта сравнения типов недопустимы: `String` с `Integer` и `Datetime` с `Time`.

```
pl.select(
    bool_num=pl.lit(True) > 0,
    time_time=pl.time(23, 58) > pl.time(0, 0),
    datetime_date=pl.datetime(1969, 7, 21, 2, 56) < pl.date(1976, 7, 20),
    str_num=pl.lit("5") < pl.lit(3).cast(pl.String), ❶
    datetime_time=pl.datetime(1999, 1, 1).dt.time() != pl.time(0, 0), ❷
).transpose( ❸
    include_header=True, header_name="comparison", column_names=["allowed"]
)
```

❶ Вы не можете непосредственно сравнивать строки и числа. Выход может состоять в предварительном приведении типов данных.

❷ Типы данных `Datetime` и `Time` также нельзя сравнивать напрямую. Но можно сначала извлечь компоненту времени из значения типа `Datetime` с помощью метода `Expr.dt.time()`.

❸ Транспонируем датафрейм для наглядности результатов сравнения.

Вывод:

```
shape: (5, 2)
┌──────────────┬─────────┐
│ comparison   ┆ allowed │
│ ---          ┆ ---     │
│ str          ┆ bool    │
╞══════════════╪═════════╡
│ bool_num     ┆ true    │
│ time_time    ┆ true    │
│ datetime_date┆ true    │
│ str_num      ┆ false   │
│ datetime_time┆ false   │
└──────────────┴─────────┘
```

# Операторы булевой алгебры

В предыдущем разделе мы комбинировали два выражения сравнения для проверки вхождения значения `x` в заданный диапазон. Воспользуемся этим примером снова, но присвоим переменной `x` значение 7. Также присвоим результаты произведенных сравнений двум переменным: `p` и `q`:

```
x = 7
p = pl.lit(3) < pl.lit(x)  # True
q = pl.lit(x) < pl.lit(5)  # False
pl.select(p & q).item()
```

Вывод:

```
False
```

Мы скомбинировали выражения `p` и `q` при помощи оператора логического И (`&`), который возвращает значение `True` только в том случае, если оба операнда равны `True`. Поскольку в нашем случае выражение `q` ложно, мы получили значение `False`. Эта булева операция называется *конъюнкцией* (conjunction).

Конъюнкция принадлежит к тройке наиболее популярных операций булевой алгебры. Двумя другими являются *дизъюнкция* (disjunction – `|`) и *отрицание* (negation – `~`)[1]. С помощью этих базовых операций можно создавать вторичные булевы операторы. В Polars реализован один такой оператор, а именно исключающее ИЛИ (`^`). В табл. 9.3 приведены все четыре булевы операции вместе с соответствующими встраиваемыми операторами и методами.

[1] Поскольку операция отрицания работает только с одним значением, формально она не относится к разделу комбинирования выражений, но мы решили рассказать о ней здесь, что наиболее логично.

***Таблица 9.3. Булевы операции в Polars***

| Операция | Оператор | Метод | Описание |
|---|---|---|---|
| Конъюнкция | `&` | *`Expr.and_()`* | Логическое И |
| Дизъюнкция | `\|` | *`Expr.or_()`* | Логическое ИЛИ |
| Отрицание | `~` | *`Expr.not_()`* | Логическое НЕ |
| Исключающее ИЛИ | `^` | *`Expr.xor()`* | Логический оператор XOR |

**Уродливые подчеркивания**

В языке Python ключевые слова `and`, `or` и `not` являются зарезервированными, что не позволяет библиотеке Polars использовать их в качестве имен переменных или методов. Именно с этим связан тот факт, что имена первых трех методов в табл. 9.3 оканчиваются на символ подчеркивания. В результате получились не очень приглядные названия методов. Так или иначе, вы все равно наверняка в большинстве случаев будете использовать вместо них встраиваемые операторы `&`, `|` и `~` соответственно.

Операция конъюнкции дает результат `True` только в случае, если оба операнда равны `True`. В следующем фрагменте кода мы применим шесть различных булевых операций ко всем возможным комбинациям `p` и `q`: к четырем операциям из табл. 9.3 мы добавим две дополнительные: *`NAND`* и *`NOR`*. Результат выполнения этого кода даст нам так называемую *таблицу истинности* (truth table):

```
(
    pl.DataFrame(
        {"p": [True, True, False, False], "q": [True, False, True, False]}
    ).with_columns(
        (pl.col("p") & pl.col("q")).alias("p & q"),
        (pl.col("p") | pl.col("q")).alias("p | q"),
        (~pl.col("p")).alias("~p"),
        (pl.col("p") ^ pl.col("q")).alias("p ^ q"),
        (~(pl.col("p") & pl.col("q"))).alias("p ↑ q"), ❶
        ((pl.col("p").or_(pl.col("q"))).not_()).alias("p ↓ q"), ❷
    )
)
```

❶ В Polars оператор NAND (NOT AND) не реализован, но его можно получить, скомбинировав операторы NOT (`~`) и AND (`&`).

❷ То же справедливо и для оператора NOR (NOT OR). Здесь мы воспользовались альтернативным синтаксисом с применением методов вместо встраиваемых операторов.

Вывод:

```
shape: (4, 8)
┌───────┬───────┬───────┬───────┬───────┬───────┬───────┬───────┐
│ p     ┆ q     ┆ p & q ┆ p | q ┆ ~p    ┆ p ^ q ┆ p ↑ q ┆ p ↓ q │
│ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   ┆ ---   │
│ bool  ┆ bool  ┆ bool  ┆ bool  ┆ bool  ┆ bool  ┆ bool  ┆ bool  │
╞═══════╪═══════╪═══════╪═══════╪═══════╪═══════╪═══════╪═══════╡
│ true  ┆ true  ┆ true  ┆ true  ┆ false ┆ false ┆ false ┆ false │
│ true  ┆ false ┆ false ┆ true  ┆ false ┆ true  ┆ true  ┆ false │
```

```
| false | true  | false | true  | true  | true  | true  | false |
| false | false | false | false | true  | false | true  | true  |
└───────┴───────┴───────┴───────┴───────┴───────┴───────┴───────┘
```

Комбинирование булевых выражений посредством булевых операций позволяет описывать сложные зависимости между выражениями. В следующем разделе мы применим те же методы и встраиваемые операции к целочисленным, а не булевым значениям.

# Битовые операции

Вы также можете применять операции AND (`&`), OR (`|`), XOR (`^`) и NOT(`~`) и к целочисленным значениям. В этом случае эти операции будут выполняться побитово[1].

Ниже показан пример применения битового оператора OR (`|`) к значениям 10 и 34, что в результате дает нам число 42[2]:

```
pl.select(pl.lit(10) | pl.lit(34)).item()
```

Вывод:

```
42
```

Под капотом Polars применяет оператор OR к каждой паре битов, составляющих числа 10 и 34. Исходящий бит будет равен 1, только если хотя бы один из входящих битов был равен 1:

```
   00001010 (десятичное число 10)
OR 00100010 (десятичное число 34)
=  00101010 (десятичное число 42)
```

Таким образом, 10 | 34 = 42, что объясняется тем, что в исходном битовом представлении второй, четвертый и шестой биты хотя бы в одном из этих чисел равняются 1. Вы можете думать об этих битах как о последовательностях булевых значений – логика здесь такая же.

В табл. 9.4 представлены четыре битовые операции с соответствующими методами и встраиваемыми операторами.

***Таблица 9.4. Битовые операции в Polars***

| Оператор | Метод | Описание |
|---|---|---|
| *&* | *Expr.and_()* | Побитовое И |
| *\|* | *Expr.or_()* | Побитовое ИЛИ |
| *~* | *Expr.not_()* | Побитовое НЕ |
| *^* | *Expr.xor()* | Побитовое XOR |

[1] Конечно, побитовые операции используются крайне редко, но ведь книга не зря называется «Python Polars: подробное руководство».

[2] Все подробности в романе Дугласа Адамса «Автостопом по Галактике».

В следующем фрагменте кода мы применим битовые операции к целочисленным столбцам:

```
bits = pl.DataFrame(
    {"x": [1, 1, 0, 0, 7, 10], "y": [1, 0, 1, 0, 2, 34]},
    schema={"x": pl.UInt8, "y": pl.UInt8}, ❶
).with_columns(
    (pl.col("x") & pl.col("y")).alias("x & y"),
    (pl.col("x") | pl.col("y")).alias("x | y"),
    (~pl.col("x")).alias("~x"),
    (pl.col("x") ^ pl.col("y")).alias("x ^ y"),
)

bits
```

❶ Мы воспользовались 8-битным беззнаковым целочисленным типом данных. Вы можете применять битовые операции к любым типам.

Вывод:

```
shape: (6, 6)
┌─────┬─────┬───────┬───────┬─────┬───────┐
│ x   ┆ y   ┆ x & y ┆ x | y ┆ ~x  ┆ x ^ y │
│ --- ┆ --- ┆ ---   ┆ ---   ┆ --- ┆ ---   │
│ u8  ┆ u8  ┆ u8    ┆ u8    ┆ u8  ┆ u8    │
╞═════╪═════╪═══════╪═══════╪═════╪═══════╡
│ 1   ┆ 1   ┆ 1     ┆ 1     ┆ 254 ┆ 0     │
│ 1   ┆ 0   ┆ 0     ┆ 1     ┆ 254 ┆ 1     │
│ 0   ┆ 1   ┆ 0     ┆ 1     ┆ 255 ┆ 1     │
│ 0   ┆ 0   ┆ 0     ┆ 0     ┆ 255 ┆ 0     │
│ 7   ┆ 2   ┆ 2     ┆ 7     ┆ 248 ┆ 5     │
│ 10  ┆ 34  ┆ 2     ┆ 42    ┆ 245 ┆ 40    │
└─────┴─────┴───────┴───────┴─────┴───────┘
```

Давайте взглянем на двоичное строковое представление этих целочисленных значений для понимания того, как работают эти операторы:

```
bits.select(pl.all().map_elements("{0:08b}".format, return_dtype=pl.String))
```

Вывод:

```
shape: (6, 6)
┌──────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
│ x        ┆ y        ┆ x & y    ┆ x | y    ┆ ~x       ┆ x ^ y    │
│ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      ┆ ---      │
│ str      ┆ str      ┆ str      ┆ str      ┆ str      ┆ str      │
╞══════════╪══════════╪══════════╪══════════╪══════════╪══════════╡
│ 00000001 ┆ 00000001 ┆ 00000001 ┆ 00000001 ┆ 11111110 ┆ 00000000 │
│ 00000001 ┆ 00000000 ┆ 00000000 ┆ 00000001 ┆ 11111110 ┆ 00000001 │
│ 00000000 ┆ 00000001 ┆ 00000000 ┆ 00000001 ┆ 11111111 ┆ 00000001 │
│ 00000000 ┆ 00000000 ┆ 00000000 ┆ 00000000 ┆ 11111111 ┆ 00000000 │
│ 00000111 ┆ 00000010 ┆ 00000010 ┆ 00000111 ┆ 11111000 ┆ 00000101 │
│ 00001010 ┆ 00100010 ┆ 00000010 ┆ 00101010 ┆ 11110101 ┆ 00101000 │
└──────────┴──────────┴──────────┴──────────┴──────────┴──────────┘
```

**Единицы и нули**

При использовании единиц и нулей для представления булевых значений результат будет таким же, за исключением оператора NOT. Инвертирование значения `True` дает `False`, тогда как инвертирование 1 дает 254 (а не 0), поскольку семь левых битов, установленных в единицу, дают в сумме число 254 (128 + 64 + 32 + 16 + 8 + 4 + 2 = 254).

Кроме того, булевы значения более эффективны в сравнении с целочисленными в плане использования памяти. И это понятно – для их хранения требуется один бит, а не восемь.

Мы всегда рекомендуем применять булевы значения, когда выражение или столбец может принимать только два значения.

# Использование функций

В табл. 9.5 перечислены все функции уровня модуля, позволяющие объединять существующие выражения в одно.

***Таблица 9.5. Функции уровня модуля для комбинирования выражений***

| Функция | Описание |
|---|---|
| *pl.all_horizontal()* | Применяет операцию побитового И горизонтально ко всем столбцам |
| *pl.any_horizontal()* | Применяет операцию побитового ИЛИ горизонтально ко всем столбцам |
| *pl.arctan2()* | Вычисляет арктангенс с двумя аргументами в радианах |
| *pl.arctan2d()* | Вычисляет арктангенс с двумя аргументами в градусах |
| *pl.arg_sort_by()* | Возвращает индексы строк, с помощью которых можно отсортировать столбцы |
| *pl.arg_where()* | Возвращает индексы, в которых условие вычисляется в `True` |
| *pl.coalesce()* | Перебирает столбцы слева направо, возвращая первое непропущенное значение |
| *pl.concat_list()* | Объединяет значения столбцов по горизонтали в один столбец типа `List` |
| *pl.concat_str()* | Объединяет значения столбцов по горизонтали в один столбец типа `String` |
| *pl.corr()* | Вычисляет корреляцию Пирсона или Спирмена между двумя столбцами |
| *pl.cov()* | Вычисляет ковариацию между двумя столбцами/выражениями |
| *pl.cum_fold()* | Накопительно сворачивает значения по всем столбцам слева направо |
| *pl.cum_reduce()* | Накопительно редуцирует значения по всем столбцам слева направо |
| *pl.cum_sum_horizontal()* | Накопительно суммирует значения по всем столбцам слева направо |
| *pl.fold()* | Накапливает значения по нескольким столбцам по горизонтали / по строкам слева направо |
| *pl.format()* | Форматирует выражения в виде строк |

***Таблица 9.5*** *(окончание)*

| Функция | Описание |
|---|---|
| *pl.map_batches()* | Применяет пользовательскую функцию к нескольким столбцам/выражениям |
| *pl.max_horizontal()* | Извлекает максимальное значение по горизонтали по всем столбцам |
| *pl.min_horizontal()* | Извлекает минимальное значение по горизонтали по всем столбцам |
| *pl.reduce()* | Редуцирует значения по нескольким столбцам по горизонтали / по строкам слева направо |
| *pl.rolling_corr()* | Вычисляет скользящую корреляцию между двумя столбцами/выражениями |
| *pl.rolling_cov()* | Вычисляет скользящую ковариацию между двумя столбцами/выражениями |
| *pl.struct()* | Собирает столбцы в один столбец с типом `Struct` |
| *pl.sum_horizontal()* | Суммирует значения по горизонтали по всем столбцам |
| *pl.when()* | Выполняет выражение when–then–otherwise |

Разумеется, мы не сможем обсудить все приведенные функции подробно, но все же приведем несколько полезных примеров. В первом мы продемонстрируем две функции для комбинирования значений нескольких выражений в одну структуру. Функции `pl.concat_list()` и `pl.struct()` служат для создания столбцов типа `List` и `Struct` соответственно. Более детально эти типы мы обсудим в главе 12. Мы будем применять функции к датафрейму, показанному ниже:

```
scientists = pl.DataFrame(
    {
        "first_name": ["George", "Grace", "John", "Kurt", "Ada"],
        "last_name": ["Boole", "Hopper", "Tukey", "Gödel", "Lovelace"],
        "country": [
            "England",
            "United States",
            "United States",
            "Austria-Hungary",
            "England",
        ],
    }
)
scientists
```

Вывод:

```
shape: (5, 3)
┌────────────┬────────────┬────────────────┐
│ first_name ┆ last_name  ┆ country        │
│ ---        ┆ ---        ┆ ---            │
│ str        ┆ str        ┆ str            │
╞════════════╪════════════╪════════════════╡
│ George     ┆ Boole      ┆ England        │
│ Grace      ┆ Hopper     ┆ United States  │
```

```
| John       | Tukey     | United States   |
| Kurt       | Gödel     | Austria-Hungary |
| Ada        | Lovelace  | England         |
└────────────┴───────────┴─────────────────┘
```

```
scientists.select(
    concat_list=pl.concat_list(pl.col("^*_name$")),
    struct=pl.struct(pl.all()),
)
```

Вывод:

```
shape: (5, 2)
┌─────────────────┬─────────────────────────────────────┐
| concat_list     | struct                              |
| ---             | ---                                 |
| list[str]       | struct[3]                           |
╞═════════════════╪═════════════════════════════════════╡
| ["George", "Boole"] | {"George","Boole","England"}       |
| ["Grace", "Hopper"] | {"Grace","Hopper","United States"} |
| ["John", "Tukey"]   | {"John","Tukey","United States"}   |
| ["Kurt", "Gödel"]   | {"Kurt","Gödel","Austria-Hungary"} |
| ["Ada", "Lovelace"] | {"Ada","Lovelace","England"}       |
└─────────────────┴─────────────────────────────────────┘
```

Здесь мы в столбце с именем `concat_list` собрали в виде списков значения исходных столбцов, имена которых удовлетворяют регулярному выражению `"^*_name$"`, а это столбцы `first_name` и `last_name`. В столбце с именем `struct` мы в виде структур объединили значения из всех столбцов, применив выражение `pl.all()`.

Теперь воспользуемся функциями `pl.concat_str()` и `pl.format()` для объединения значений столбцов в строковые представления. Вторая функция дает нам больше гибкости в плане объединения столбцов:

```
scientists.select(
    concat_str=pl.concat_str(pl.all(), separator=" "),
    format=pl.format("{}, {} from {}", "last_name", "first_name", "country"),
)
```

Вывод:

```
shape: (5, 2)
┌──────────────────────────┬──────────────────────────────────┐
| concat_str               | format                           |
| ---                      | ---                              |
| str                      | str                              |
╞══════════════════════════╪══════════════════════════════════╡
| George Boole England     | Boole, George from England       |
| Grace Hopper United States | Hopper, Grace from United States |
| John Tukey United States | Tukey, John from United States   |
| Kurt Gödel Austria-Hungary | Gödel, Kurt from Austria-Hungary |
| Ada Lovelace England     | Lovelace, Ada from England       |
└──────────────────────────┴──────────────────────────────────┘
```

Функции `pl.all_horizontal()` и `pl.any_horizontal()` аналогичны операциям И (`&`) и ИЛИ (`|`) применительно к нескольким столбцам. Они бывают особенно полезны, если вам необходимо объединить множество столбцов и вы не хотите перечислять все их имена:

```
prefs = pl.DataFrame(
    {
        "id": [1, 7, 42, 101, 999],
        "has_pet": [True, False, True, False, True],
        "likes_travel": [False, False, False, False, True],
        "likes_movies": [True, False, True, False, True],
        "likes_books": [False, False, True, True, True],
    }
).with_columns(
    all=pl.all_horizontal(pl.exclude("id")),
    any=pl.any_horizontal(pl.exclude("id")),
)

prefs
```

Вывод:

```
shape: (5, 7)
┌─────┬────────┬─────────────┬──────────────┬─────────────┬───────┬───────┐
│ id  ┆ has_pet┆ likes_travel┆ likes_movies ┆ likes_books ┆ all   ┆ any   │
│ --- ┆ ---    ┆ ---         ┆ ---          ┆ ---         ┆ ---   ┆ ---   │
│ i64 ┆ bool   ┆ bool        ┆ bool         ┆ bool        ┆ bool  ┆ bool  │
╞═════╪════════╪═════════════╪══════════════╪═════════════╪═══════╪═══════╡
│ 1   ┆ true   ┆ false       ┆ true         ┆ false       ┆ false ┆ true  │
│ 7   ┆ false  ┆ false       ┆ false        ┆ false       ┆ false ┆ false │
│ 42  ┆ true   ┆ false       ┆ true         ┆ true        ┆ false ┆ true  │
│ 101 ┆ false  ┆ false       ┆ false        ┆ true        ┆ false ┆ true  │
│ 999 ┆ true   ┆ true        ┆ true         ┆ true        ┆ true  ┆ true  │
└─────┴────────┴─────────────┴──────────────┴─────────────┴───────┴───────┘
```

Функции `pl.sum_horizontal()`, `pl.max_horizontal()` и `pl.min_horizontal()` служат для вычисления суммы, максимума и минимума по столбцам соответственно. Их можно применять к числовым и булевым столбцам:

```
prefs.select(
    sum=pl.sum_horizontal(pl.all()),
    max=pl.max_horizontal(pl.all()),
    min=pl.min_horizontal(pl.all()),
)
```

Вывод:

```
shape: (5, 3)
┌─────┬─────┬─────┐
│ sum ┆ max ┆ min │
│ --- ┆ --- ┆ --- │
│ i64 ┆ i64 ┆ i64 │
╞═════╪═════╪═════╡
```

```
| 4    | 1   | 0   |
| 7    | 7   | 0   |
| 46   | 42  | 0   |
| 103  | 101 | 0   |
| 1005 | 999 | 1   |
└──────┴─────┴─────┘
```

## When, Then, Otherwise

Функция `pl.when()` достойна отдельного упоминания. С помощью нее можно создавать условные выражения. Вы можете думать об этой функции как о векторизованном варианте инструкции `if`. Ниже приведен пример ее использования:

```
prefs.select(
    pl.col("id"),
    likes_what=pl.when(pl.all_horizontal(pl.col("^likes_.*$")))
    .then(pl.lit("Likes everything"))
    .when(pl.any_horizontal(pl.col("^likes_.*$")))
    .then(pl.lit("Likes something"))
    .otherwise(pl.lit("Likes nothing")),
)
```

Вывод:

```
shape: (5, 2)
┌─────┬──────────────────┐
│ id  ┆ likes_what       │
│ --- ┆ ---              │
│ i64 ┆ str              │
╞═════╪══════════════════╡
│ 1   ┆ Likes something  │
│ 7   ┆ Likes nothing    │
│ 42  ┆ Likes something  │
│ 101 ┆ Likes something  │
│ 999 ┆ Likes everything │
└─────┴──────────────────┘
```

Функция `pl.when()` является исключительно мощной. С ее помощью вы можете создавать сложные условные выражения с помощью одной инструкции. В показанном выше примере мы очень быстро и гибко проверили значения в столбцах `likes_travel`, `likes_movies` и `likes_books` и сделали на их основе выводы о том, кто любит все, кто – ничего, а кто – кое-что. И все это с помощью одной функции `pl.when()`.

Вычисляется эта функция подобно конструкции `if-elif-else` в Python. То есть берется первое вхождение значения `True`, а остальные проверки не выполняются. Если ни одно из условий не вычисляется в `True`, применяется блок `pl.otherwise()`, если он включен в выражение. В противном случае возвращается значение `null`.

## Что еще?

Как мы упомянули, если в конструкцию функции `pl.when()` не включить блок `otherwise()`, будет возвращено значение `null`. В следующем примере мы создадим датафрейм с данными о платежах. Если сумма платежа превышает 1000, необходимо как-то пометить операцию для будущей проверки:

```
orders = pl.DataFrame(
    {
        "order_amount": [500, 750, 1200, 800, 1100],
        "status": [
            "Approved",
            "Processing",
            "Processing",
            "Declined",
            "Processing",
        ],
    }
)
orders.with_columns(
    status=pl.when(pl.col("order_amount") > 1000).then(pl.lit("Flagged"))
)
```

Вывод:

```
shape: (5, 2)
┌──────────────┬──────────┐
│ order_amount ┆ status   │
│ ---          ┆ ---      │
│ i64          ┆ str      │
╞══════════════╪══════════╡
│ 500          ┆ null     │
│ 750          ┆ null     │
│ 1200         ┆ Flagged  │
│ 800          ┆ null     │
│ 1100         ┆ Flagged  │
└──────────────┴──────────┘
```

Проблема в том, что в результате исходные статусы перезаписались значениями `null`. Это может быть не слишком удобно. В Python, к примеру, отсутствие блока `else` оставляет значения неизменными. Если вам необходимо оставить исходные значения при невыполнении условий, вы можете включить блок `otherwise` с указанием исходного столбца в качестве аргумента:

```
orders.with_columns(
    status=pl.when(pl.col("order_amount") > 1000)
    .then(pl.lit("Flagged"))
    .otherwise(pl.col("status"))
)
```

Вывод:

```
shape: (5, 2)
┌──────────────┬────────────┐
│ order_amount ┆ status     │
```

```
| ---          | ---        |
| i64          | str        |
╞══════════════╪════════════╡
| 500          | Approved   |
| 750          | Processing |
| 1200         | Flagged    |
| 800          | Declined   |
| 1100         | Flagged    |
└──────────────┴────────────┘
```

Согласитесь, так намного лучше.

Подробности использования других функций можно почерпнуть из официальной документации.

# Заключение

В этой главе вы узнали, что:

- комбинирование выражений в Polars позволяет выполнять сложные манипуляции с данными на основе нескольких значений или столбцов;
- вы можете использовать для комбинирования выражений как встраиваемые операторы, такие как `+`, `-`, `*`, `/`, `&`, `|` и другие, так и цепочки методов. Оба способа дадут одинаковый результат;
- арифметические операции можно выполнять с помощью встраиваемых операторов или соответствующих методов, что позволяет складывать, вычитать, перемножать или делить выражения;
- операции сравнения в Polars не могут объединяться в цепочки, подобно тому, как это происходит в Python. Вместо этого вам необходимо комбинировать несколько операций сравнения при помощи булевых операторов, таких как И (`&`) и ИЛИ (`|`);
- для комбинирования булевых выражений можно применять операции булевой алгебры, такие как конъюнкция (`&`), дизъюнкция (`|`) и отрицание (`^`);
- битовые операции могут быть применены к целочисленным выражениям при помощи тех же операторов и будут вычисляться на уровне битов;
- в Polars присутствует большое количество функций уровня модуля для комбинирования выражений, таких как `pl.concat_list()`, `pl.concat_str()`, `pl.format()`, `pl.all_horizontal()`, `pl.any_horizontal()`, `pl.sum_horizontal()` и `pl.when()`. Все они позволяют осуществлять сложные манипуляции с данными.

На этом мы завершаем третью часть книги, посвященную выражениям. Теперь вы знаете, как можно создавать, расширять и комбинировать выражения в Polars.

# ЧАСТЬ IV

# Преобразования

# Глава 10

# Выбор и создание столбцов

Теперь, когда вы досконально понимаете, как функционируют выражения, давайте посмотрим, как можно их применять на практике. В этой главе мы рассмотрим операции по работе со столбцами датафреймов[1]. Мы сосредоточимся на выборе существующих столбцов и создании новых, что составляет основную часть работы при анализе данных.

Сначала мы вспомним, как работать с методом датафреймов `select()`, с которым уже встречались в главе 7. После этого познакомимся с более универсальным и гибким способом выбора столбцов, связанным с применением так называемых селекторов столбцов. С помощью селекторов можно указать нужные нам столбцы посредством разных критериев, включая имена столбцов, их типы данных и расположение в датафрейме. Также селекторы можно комбинировать различными способами. После этого мы узнаем, как создавать новые столбцы и менять их положение. Наконец, мы кратко коснемся операций по работе со столбцами, таких как переименование, удаление и объединение столбцов из разных датафреймов.

На протяжении этой главы мы будем работать с датафреймом, посвященным нашим любимым персонажам вселенной «Звездных войн»:

```
starwars = pl.read_parquet("data/starwars.parquet")
rebels = starwars.drop("films").filter(
    pl.col("name").is_in(["Luke Skywalker", "Leia Organa", "Han Solo"])
)

print(rebels[:, :6]) ❶
print(rebels[:, 6:11])
print(rebels[:, 11:])
```

❶ Мы разбили наш датафрейм на столбцы для более удобного представления на страницах книги.

[1] Все рассмотренные операции применимы и к ленивым датафреймам.

Вывод:

```
shape: (3, 6)
┌──────────────┬────────┬──────┬────────────┬────────────┬───────────┐
│ name         ┆ height ┆ mass ┆ hair_color ┆ skin_color ┆ eye_color │
│ ---          ┆ ---    ┆ ---  ┆ ---        ┆ ---        ┆ ---       │
│ str          ┆ u16    ┆ f64  ┆ str        ┆ str        ┆ str       │
╞══════════════╪════════╪══════╪════════════╪════════════╪═══════════╡
│ Han Solo     ┆ 180    ┆ 80.0 ┆ brown      ┆ fair       ┆ brown     │
│ Leia Organa  ┆ 150    ┆ 49.0 ┆ brown      ┆ light      ┆ brown     │
│ Luke Skywalker ┆ 172  ┆ 77.0 ┆ blond      ┆ fair       ┆ blue      │
└──────────────┴────────┴──────┴────────────┴────────────┴───────────┘
shape: (3, 5)
┌────────────┬────────┬───────────┬────────────┬─────────┐
│ birth_year ┆ sex    ┆ gender    ┆ homeworld  ┆ species │
│ ---        ┆ ---    ┆ ---       ┆ ---        ┆ ---     │
│ f64        ┆ cat    ┆ cat       ┆ str        ┆ str     │
╞════════════╪════════╪═══════════╪════════════╪═════════╡
│ 29.0       ┆ male   ┆ masculine ┆ Corellia   ┆ Human   │
│ 19.0       ┆ female ┆ feminine  ┆ Alderaan   ┆ Human   │
│ 19.0       ┆ male   ┆ masculine ┆ Tatooine   ┆ Human   │
└────────────┴────────┴───────────┴────────────┴─────────┘
shape: (3, 4)
┌──────────────────────┬──────────────────────┬────────────┬──────────────┐
│ vehicles             ┆ starships            ┆ birth_date ┆ screen_time  │
│ ---                  ┆ ---                  ┆ ---        ┆ ---          │
│ list[str]            ┆ list[str]            ┆ date       ┆ duration[µs] │
╞══════════════════════╪══════════════════════╪════════════╪══════════════╡
│ null                 ┆ ["Millennium Falcon"… ┆ 1948-06-01 ┆ 1h 12m 37s   │
│ ["Imperial Speeder B… ┆ null                 ┆ 1958-05-30 ┆ 1h 3m 40s    │
│ ["Snowspeeder", "Imp… ┆ ["X-wing", "Imperial… ┆ 1958-05-30 ┆ 1h 58m 44s   │
└──────────────────────┴──────────────────────┴────────────┴──────────────┘
```

В датафрейме `rebels` присутствуют три строки и 15 столбцов различных типов, чего будет достаточно для демонстрации всех необходимых операций для работы со столбцами. Этот датафрейм был взят из пакета языка R dplyr. Мы добавили в него столбцы `birth_date` (дата рождения) и `screen_time` (экранное время)[1].

Все инструкции по загрузке нужных файлов и установке пакетов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

---

[1] Годы рождения в столбце `birth_year` представлены в летоисчислении до Битвы при Явине в стандартном галактическом календаре, что означает, что Люку и Лее было по 19 лет во время этого события. В то же время в столбце `birth_date` даты представлены в летоисчислении от Рождества Христова в соответствии с григорианским календарем, если за дату Битвы при Явине взять 25 мая 1977 года, т. е. дату выхода фильма «Звездные войны. Эпизод 4: Новая надежда» в США. В столбце `screen_time` представлено общее экранное время для персонажей во всех трех фильмах оригинальной трилогии, взятое с сайта *https://www.screentimecentral.com/star-wars-characters*. В общем, мы изрядно повеселились, собирая информацию для этих двух новых столбцов в датафрейме.

# Выбор столбцов

Столбцы из датафрейма можно выбрать при помощи метода *select()*. При этом вы можете извлечь один, несколько или все столбцы в датафрейме в произвольном порядке. Более того, один столбец вы можете извлечь несколько раз, если обеспечите им разные имена (имена столбцов должны быть уникальными). Также вы можете воспользоваться этим методом и для создания новых столбцов, как мы увидим далее. На рис. 10.1 проиллюстрирована концепция операции выбора столбцов.

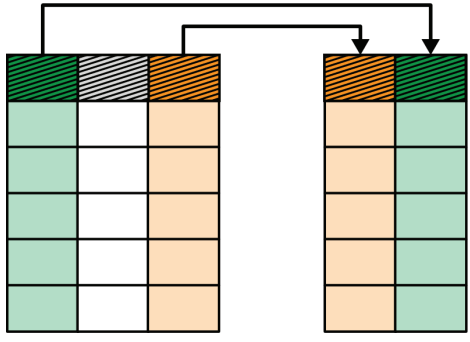


**Рис. 10.1** ❖ Выбор столбцов осуществляется по их именам, типам данных или позиции

Выбор столбцов бывает особенно удобен при наличии в исходном датафрейме большого количества колонок, некоторые из которых вам не нужны.

Для выбора столбцов необходимо указать, какие именно столбцы следует сохранить. В главе 7 мы рассмотрели два способа выполнить это действие: с помощью строкового представления имен столбцов и посредством функции `pl.col()`. Первый способ короче и лаконичнее, тогда как второй задействует выражение, что позволяет преобразовать столбец при необходимости. Напомним, как действуют оба способа:

```
rebels.select(
    "name",
    pl.col("homeworld"),
    pl.col("^.*_color$"),
    (pl.col("height") / 100).alias("height_m"),
)
```

Вывод:

```
┌──────────────────┬────────────┬────────────┬────────────┬───────────┬──────────┐
│ name             ┆ homeworld  ┆ hair_color ┆ skin_color ┆ eye_color ┆ height_m │
│ ---              ┆ ---        ┆ ---        ┆ ---        ┆ ---       ┆ ---      │
│ str              ┆ str        ┆ str        ┆ str        ┆ str       ┆ f64      │
╞══════════════════╪════════════╪════════════╪════════════╪═══════════╪══════════╡
```

```
| Han Solo       | Corellia  | brown      | fair       | brown      | 1.8      |
| Leia Organa    | Alderaan  | brown      | light      | brown      | 1.5      |
| Luke Skywalker | Tatooine  | blond      | fair       | blue       | 1.72     |
```

Обратите внимание, что в результате столбцы появляются в порядке, в котором вы их указали.

В следующем разделе мы познакомимся с третьим способом выбора столбцов – при помощи селекторов.

## Знакомство с селекторами

Лучший способ описать *селекторы* (selector) состоит в демонстрации их работы. Для применения селекторов вам сперва необходимо импортировать модуль polars.selectors, как показано ниже:

```
import polars.selectors as cs
```

Принято импортировать этот модуль с алиасом cs, означающим *селекторы столбцов* (column selectors). В этом модуле присутствует более 30 функций, которые можно использовать для указания нужных вам столбцов.

**Селекторы повсюду**

Селекторы могут быть использованы не только с методом `select()`, но и везде, где необходимо перечислить нужные столбцы, включая методы `filter()` и `group_by()`.

Функцией, во многом напоминающей функцию `pl.col()`, является *`cs.by_name()`*. На самом деле под капотом селекторы представляют собой выражения, что позволяет использовать их так же точно, как и выражения. Таким образом, наш предыдущий пример можно переписать следующим образом:

```
rebels.select(
    "name",
    cs.by_name("homeworld"),
    cs.by_name("^.*_color$"),
    (cs.by_name("height") / 100).alias("height_m"),
)
```

Вывод:

```
shape: (3, 6)
┌────────────────┬────────────┬────────────┬────────────┬────────────┬──────────┐
| name           | homeworld  | hair_color | skin_color | eye_color  | height_m |
| ---            | ---        | ---        | ---        | ---        | ---      |
| str            | str        | str        | str        | str        | f64      |
╞════════════════╪════════════╪════════════╪════════════╪════════════╪══════════╡
| Han Solo       | Corellia   | brown      | fair       | brown      | 1.8      |
| Leia Organa    | Alderaan   | brown      | light      | brown      | 1.5      |
| Luke Skywalker | Tatooine   | blond      | fair       | blue       | 1.72     |
└────────────────┴────────────┴────────────┴────────────┴────────────┴──────────┘
```

Эта функция сама по себе никак не оправдывает применение селекторов. Она даже длиннее, чем функция `pl.col()`. Гибкость селекторов проявляется при использовании других функций и их комбинировании.

В следующих трех разделах мы взглянем на три типа селекторов, позволяющих обращаться к столбцам по именам, типам данных и позиции.

# Выбор столбцов по имени

Помимо `cs.by_name()`, существует еще несколько функций, позволяющих выбрать столбцы по имени. Все они перечислены в табл. 10.1.

***Таблица 10.1. Функции для выбора столбцов по имени***

| Функция | Описание |
|---|---|
| *cs.alpha()* | Выбирает все столбцы с буквенными именами (т. е. содержащими только буквы) |
| *cs.alphanumeric()* | Выбирает все столбцы с буквенно-цифровыми именами (т. е. содержащими только буквы и цифры от 0 до 9) |
| *cs.by_name()* | Выбирает все столбцы, соответствующие заданным именам |
| *cs.contains()* | Выбирает все столбцы, в именах которых содержатся заданные буквенные подстроки |
| *cs.digit()* | Выбирает все столбцы, имена которых состоят только из цифр |
| *cs.ends_with()* | Выбирает все столбцы, имена которых оканчиваются на заданную подстроку |
| *cs.matches()* | Выбирает все столбцы, имена которых соответствуют заданному шаблону регулярных выражений |
| *cs.starts_with()* | Выбирает все столбцы, имена которых начинаются на заданную подстроку |

Ниже мы приведем несколько примеров. Для выбора всех столбцов, имена которых начинаются с `birth_`, можно воспользоваться функцией `cs.starts_with()` следующим образом:

```
rebels.select(cs.starts_with("birth_"))
```

Вывод:

```
shape: (3, 2)
┌────────────┬────────────┐
│ birth_year ┆ birth_date │
│ ---        ┆ ---        │
│ f64        ┆ date       │
╞════════════╪════════════╡
│ 29.0       ┆ 1948-06-01 │
│ 19.0       ┆ 1958-05-30 │
│ 19.0       ┆ 1958-05-30 │
└────────────┴────────────┘
```

А для выбора столбцов с именами, оканчивающимися на `_color`, подойдет функция `cs.ends_with()`:

```
rebels.select(cs.ends_with("_color"))
```

Вывод:

```
shape: (3, 3)
┌────────────┬────────────┬───────────┐
│ hair_color ┆ skin_color ┆ eye_color │
│ ---        ┆ ---        ┆ ---       │
│ str        ┆ str        ┆ str       │
╞════════════╪════════════╪═══════════╡
│ brown      ┆ fair       ┆ brown     │
│ brown      ┆ light      ┆ brown     │
│ blond      ┆ fair       ┆ blue      │
└────────────┴────────────┴───────────┘
```

Выбрать только те столбцы, в именах которых присутствует символ подчеркивания, можно с помощью функции `cs.contains()`, как показано ниже:

```
rebels.select(cs.contains("_"))
```

Вывод:

```
shape: (3, 6)
┌────────────┬────────────┬───────────┬────────────┬────────────┬──────────────┐
│ hair_color ┆ skin_color ┆ eye_color ┆ birth_year ┆ birth_date ┆ screen_time  │
│ ---        ┆ ---        ┆ ---       ┆ ---        ┆ ---        ┆ ---          │
│ str        ┆ str        ┆ str       ┆ f64        ┆ date       ┆ duration[µs] │
╞════════════╪════════════╪═══════════╪════════════╪════════════╪══════════════╡
│ brown      ┆ fair       ┆ brown     ┆ 29.0       ┆ 1948-06-01 ┆ 1h 12m 37s   │
│ brown      ┆ light      ┆ brown     ┆ 19.0       ┆ 1958-05-30 ┆ 1h 3m 40s    │
│ blond      ┆ fair       ┆ blue      ┆ 19.0       ┆ 1958-05-30 ┆ 1h 58m 44s   │
└────────────┴────────────┴───────────┴────────────┴────────────┴──────────────┘
```

Функция `cs.matches()` позволит вам выбрать столбцы, воспользовавшись шаблонами регулярных выражений. К примеру, чтобы оставить только столбцы, имена которых состоят из четырех строчных букв, можно выполнить следующую инструкцию:

```
rebels.select(cs.matches("^[a-z]{4}$"))
```

Вывод:

```
shape: (3, 2)
┌──────────────┬──────┐
│ name         ┆ mass │
│ ---          ┆ ---  │
│ str          ┆ f64  │
╞══════════════╪══════╡
```

```
| Han Solo       | 80.0 |
| Leia Organa    | 49.0 |
| Luke Skywalker | 77.0 |
└────────────────┴──────┘
```

Теперь давайте перейдем к селекторам, позволяющим отбирать столбцы на основе типов данных.

## Выбор столбцов по типу данных

Бывают ситуации, когда вас интересуют столбцы с данными определенного типа. К примеру, вам может понадобиться агрегировать все числовые столбцы в датафрейме с расчетом среднего:

```
rebels.group_by("hair_color").agg(cs.numeric().mean())
```

Вывод:

```
shape: (2, 4)
┌────────────┬────────┬──────┬────────────┐
| hair_color | height | mass | birth_year |
| ---        | ---    | ---  | ---        |
| str        | f64    | f64  | f64        |
╞════════════╪════════╪══════╪════════════╡
| brown      | 165.0  | 64.5 | 24.0       |
| blond      | 172.0  | 77.0 | 19.0       |
└────────────┴────────┴──────┴────────────┘
```

Функция `cs.numeric()` выбирает столбцы с беззнаковым и знаковым целочисленным типом и типом с плавающей запятой. Похожим образом работает и функция `cs.temporal()`, но она выбирает столбцы с датами, временем и длительностями. Это бывает очень удобно. В табл. 10.2 приведены все селекторные функции подобного рода.

***Таблица 10.2. Функции для выбора столбцов по типам данных***

| Функция | Описание |
|---|---|
| *cs.binary()* | Выбирает все столбцы двоичного типа |
| *cs.boolean()* | Выбирает все столбцы булева типа |
| *cs.by_dtype()* | Выбирает все столбцы, соответствующие заданным типам данных |
| *cs.categorical()* | Выбирает все столбцы категориального типа |
| *cs.date()* | Выбирает все столбцы типа дата |
| *cs.datetime()* | Выбирает все столбцы типа дата и время с возможностью фильтрации по часовым поясам |
| *cs.decimal()* | Выбирает все столбцы десятичного типа с фиксированной запятой |
| *cs.duration()* | Выбирает все столбцы с временными интервалами с возможностью фильтрации по единице времени |

***Таблица 10.2*** *(окончание)*

| Функция | Описание |
|---|---|
| *cs.float()* | Выбирает все столбцы вещественного типа с плавающей запятой |
| *cs.integer()* | Выбирает все столбцы целочисленного типа |
| *cs.numeric()* | Выбирает все числовые столбцы |
| *cs.signed_integer()* | Выбирает все столбцы знакового целочисленного типа |
| *cs.string()* | Выбирает все столбцы строкового типа с возможностью выбора категориальных столбцов |
| *cs.temporal()* | Выбирает все столбцы временного типа |
| *cs.time()* | Выбирает все столбцы, хранящие время |
| *cs.unsigned_integer()* | Выбирает все столбцы беззнакового целочисленного типа |

Ниже приведено еще несколько примеров использования таких функций. Выбор столбцов строкового типа:

```
rebels.select(cs.string())
```

Вывод:

```
shape: (3, 6)
┌──────────────┬────────────┬────────────┬──────────┬────────────┬─────────┐
│ name         ┆ hair_color ┆ skin_color ┆ eye_color ┆ homeworld ┆ species │
│ ---          ┆ ---        ┆ ---        ┆ ---       ┆ ---       ┆ ---     │
│ str          ┆ str        ┆ str        ┆ str       ┆ str       ┆ str     │
╞══════════════╪════════════╪════════════╪══════════╪════════════╪═════════╡
│ Han Solo     ┆ brown      ┆ fair       ┆ brown     ┆ Corellia  ┆ Human   │
│ Leia Organa  ┆ brown      ┆ light      ┆ brown     ┆ Alderaan  ┆ Human   │
│ Luke Skywalker ┆ blond    ┆ fair       ┆ blue      ┆ Tatooine  ┆ Human   │
└──────────────┴────────────┴────────────┴──────────┴────────────┴─────────┘
```

Вывод столбцов с типом данных, относящимся ко времени:

```
rebels.select(cs.temporal())
```

Вывод:

```
shape: (3, 2)
┌────────────┬──────────────┐
│ birth_date ┆ screen_time  │
│ ---        ┆ ---          │
│ date       ┆ duration[μs] │
╞════════════╪══════════════╡
│ 1948-06-01 ┆ 1h 12m 37s   │
│ 1958-05-30 ┆ 1h 3m 40s    │
│ 1958-05-30 ┆ 1h 58m 44s   │
└────────────┴──────────────┘
```

Также вы можете выбрать столбцы с вложенными типами, указав при этом внутренний тип данных. Например, чтобы оставить только столбцы со списками, хранящими строки, вы можете выполнить следующую инструкцию:

```
rebels.select(cs.by_dtype(pl.List(pl.String)))
```

Вывод:

```
shape: (3, 2)
┌──────────────────────┬──────────────────────┐
│ vehicles             ┆ starships            │
│ ---                  ┆ ---                  │
│ list[str]            ┆ list[str]            │
╞══════════════════════╪══════════════════════╡
│ null                 ┆ ["Millennium Falcon"… │
│ ["Imperial Speeder B… ┆ null                 │
│ ["Snowspeeder", "Imp… ┆ ["X-wing", "Imperial… │
└──────────────────────┴──────────────────────┘
```

# Выбор столбцов по позиции

Третий тип селекторов при выборе столбцов использует их позиции в датафрейме. В табл. 10.3 показаны соответствующие функции.

***Таблица 10.3. Функции для выбора столбцов по позиции***

| Функция | Описание |
|---|---|
| *cs.by_index()* | Выбирает все столбцы, соответствующие заданным индексам (или диапазону) |
| *cs.first()* | Выбирает первый столбец в текущей области видимости |
| *cs.last()* | Выбирает последний столбец в текущей области видимости |

Эти функции требуются не так часто. Например, в вашем распоряжении может оказаться датафрейм со множеством шаблонных столбцов и трудно различимыми именами. В таком случае вы могли бы выбрать каждый третий столбец с помощью функции `cs.by_index()` и объекта `range`. В нашем примере это могло бы выглядеть так:

```
rebels.select(cs.by_index(range(0, 999, 3))) ❶
```

❶ Размер диапазона не важен, если он превышает количество столбцов в датафрейме.

Вывод:

```
shape: (3, 5)
┌─────────────┬────────────┬────────────┬────────────┬──────────────────────┐
│ name        ┆ hair_color ┆ birth_year ┆ homeworld  ┆ starships            │
│ ---         ┆ ---        ┆ ---        ┆ ---        ┆ ---                  │
│ str         ┆ str        ┆ f64        ┆ str        ┆ list[str]            │
╞═════════════╪════════════╪════════════╪════════════╪══════════════════════╡
│ Han Solo    ┆ brown      ┆ 29.0       ┆ Corellia   ┆ ["Millennium Falcon"… │
│ Leia Organa ┆ brown      ┆ 19.0       ┆ Alderaan   ┆ null                 │
│ Luke Skywalker ┆ blond   ┆ 19.0       ┆ Tatooine   ┆ ["X-wing", "Imperial… │
└─────────────┴────────────┴────────────┴────────────┴──────────────────────┘
```

Еще один пример: скажем, вы только что создали два новых столбца в датафрейме с помощью метода `with_columns()` (о нем мы поговорим далее) и теперь хотите выбрать их, а также столбец `name`. Зная, что новые столбцы были добавлены в конец датафрейма, вы можете воспользоваться следующей инструкцией:

```
rebels.select("name", cs.by_index(range(-2, 0)))
```

Вывод:

```
shape: (3, 3)
┌─────────────────┬────────────┬─────────────────┐
│ name            ┆ birth_date ┆ screen_time     │
│ ---             ┆ ---        ┆ ---             │
│ str             ┆ date       ┆ duration[μs]    │
╞═════════════════╪════════════╪═════════════════╡
│ Han Solo        ┆ 1948-06-01 ┆ 1h 12m 37s      │
│ Leia Organa     ┆ 1958-05-30 ┆ 1h 3m 40s       │
│ Luke Skywalker  ┆ 1958-05-30 ┆ 1h 58m 44s      │
└─────────────────┴────────────┴─────────────────┘
```

Если бы дело касалось одного столбца, вы могли бы применить функцию `cs.last()`.

**Выход за границы диапазона**

Если передать функции `cs.index()` индексы, выходящие за границы диапазона датафрейма, вы получите ошибку следующего содержания:

```
rebels.select(cs.by_index(20))
```

Вывод:

```
ColumnNotFoundError: nth

Resolved plan until failure:

---> FAILED HERE RESOLVING 'select' <---
DF ["name", "height", "mass", "hair_color"]; PROJECT */15 COLUMNS
```

В то же время если передать этой функции объект `range` с выходящими за пределы датафрейма индексами (`range(20, 22)`), ошибки не будет, а в результате вы получите пустой датафрейм, как показано ниже:

```
rebels.select(cs.by_index(range(20, 22)))
```

Вывод:

```
shape: (0, 0)
┌┐
╞╡
└┘
```

# Комбинирование селекторов

Селекторы можно комбинировать друг с другом для получения сложных выборок. Именно здесь проявляется их гибкость и универсальность, поскольку ранее вы не могли подобным образом указывать нужные вам столбцы.

К примеру, в главе 7 мы говорили, что никак не можем одновременно задать критерии для выбора столбцов по имени и типу данных. С селекторами возможно все! Допустим, если нам нужно выбрать из датафрейма столбец с именем `hair_color` и все числовые столбцы, мы можем записать это так:

```
rebels.select(cs.by_name("hair_color") | cs.numeric())
```

Вывод:

```
shape: (3, 4)
┌────────┬──────┬────────────┬────────────┐
│ height ┆ mass ┆ hair_color ┆ birth_year │
│ ---    ┆ ---  ┆ ---        ┆ ---        │
│ u16    ┆ f64  ┆ str        ┆ f64        │
╞════════╪══════╪════════════╪════════════╡
│ 180    ┆ 80.0 ┆ brown      ┆ 29.0       │
│ 150    ┆ 49.0 ┆ brown      ┆ 19.0       │
│ 172    ┆ 77.0 ┆ blond      ┆ 19.0       │
└────────┴──────┴────────────┴────────────┘
```

Символ `|` представляет собой логический оператор ИЛИ – один из пяти операторов, реализованных для селекторов. Все операторы показаны в табл. 10.4.

***Таблица 10.4. Операторы для комбинирования селекторов***

| Операция | Встраиваемый оператор | Описание |
|---|---|---|
| Объединение | `\|` | x или y, или оба |
| Пересечение | `&` | x и y |
| Исключение | `-` | x, но не y |
| Исключающее ИЛИ | `^` | x или y, но не оба |
| Отрицание | `~` | Не x |

В приведенном ниже фрагменте кода продемонстрированы все пять операторов на примере датафрейма `df`:

```
df = pl.DataFrame({"d": 1, "i": True, "s": True, "c": True, "o": 1.0})

print(df)

x = cs.by_name("d", "i", "s")
y = cs.boolean()

print("\nселектор => столбцы")
```

```
for s in ["x", "y", "x | y", "x & y", "x - y", "x ^ y", "~x", "x - x"]:
    print(f"{s:8} => {cs.expand_selector(df, eval(s))}")
```

Вывод:

```
shape: (1, 5)
┌─────┬──────┬──────┬──────┬─────┐
│ d   ┆ i    ┆ s    ┆ c    ┆ o   │
│ --- ┆ ---  ┆ ---  ┆ ---  ┆ --- │
│ i64 ┆ bool ┆ bool ┆ bool ┆ f64 │
╞═════╪══════╪══════╪══════╪═════╡
│ 1   ┆ true ┆ true ┆ true ┆ 1.0 │
└─────┴──────┴──────┴──────┴─────┘

селектор => столбцы
x        => ('d', 'i', 's')
y        => ('i', 's', 'c')
x | y    => ('d', 'i', 's', 'c')
x & y    => ('i', 's')
x - y    => ('d',)
x ^ y    => ('d', 'c')
~x       => ('c', 'o')
x - x    => ()
```

Хотя подобные операторы мы называем операторами для работы со множествами, в результате их применения мы на самом деле получаем кортежи. И это хорошо, поскольку позволяет сохранить порядок следования столбцов.

Применение селектора `x - x` показывает, что в результате применения оператора может вовсе не остаться выбранных столбцов, что соответствует пустому датафрейму, как видно ниже:

```
df.select(x - x)
```

Вывод:

```
shape: (0, 0)
┌┐
╞╡
└┘
```

## Перемещение столбцов в датафрейме

Ниже показан необычный фрагмент кода, позволяющий переместить указанные столбцы в начало датафрейма, что возможно благодаря оператору *:=*, появившемуся в Python версии 3.8. Этот оператор, который за внешнее сходство прозвали *моржом* (walrus), позволяет присвоить селектор переменной `first` в первом аргументе, которую вы можете повторно использовать и изменять во втором аргументе. Показанный ниже пример перемещает указанные столбцы (`c` и `i`) в начало датафрейма:

```
print(df.select(first := cs.by_name("c", "i"), ~first))
print(f"first: {first}, ~first: {~first}")
```

Вывод:

```
shape: (1, 5)
┌──────┬──────┬─────┬──────┬─────┐
│ c    ┆ i    ┆ d   ┆ s    ┆ o   │
│ ---  ┆ ---  ┆ --- ┆ ---  ┆ --- │
│ bool ┆ bool ┆ i64 ┆ bool ┆ f64 │
╞══════╪══════╪═════╪══════╪═════╡
│ true ┆ true ┆ 1   ┆ true ┆ 1.0 │
└──────┴──────┴─────┴──────┴─────┘
first: cols(["c", "i"]), ~first: selector
```

В следующем примере мы переместим последний столбец (с именем `o`) на первое место. Это может быть особенно полезно, например, после создания нового столбца:

```
print(df.select(first := cs.last(), ~first))
print(f"first: {first}, ~first: {~first}")
```

Вывод:

```
shape: (1, 5)
┌─────┬─────┬──────┬──────┬──────┐
│ o   ┆ d   ┆ i    ┆ s    ┆ c    │
│ --- ┆ --- ┆ ---  ┆ ---  ┆ ---  │
│ f64 ┆ i64 ┆ bool ┆ bool ┆ bool │
╞═════╪═════╪══════╪══════╪══════╡
│ 1.0 ┆ 1   ┆ true ┆ true ┆ true │
└─────┴─────┴──────┴──────┴──────┘
first: nth(-1), ~first: selector
```

# Создание столбцов

Создать один или несколько столбцов в датафрейме можно при помощи метода *`with_columns()`*. В результате созданные столбцы будут добавлены в конец датафрейма. На рис. 10.2 эта операция показана графически.

Вряд ли для персонажей «Звездных войн» это было актуально, но мы можем воспользоваться информацией из столбцов `mass` и `height` и рассчитать на ее основе индекс массы тела (body mass index – BMI):

```
rebels.with_columns(bmi=pl.col("mass") / ((pl.col("height") / 100) ** 2))
```

Вывод:

```
shape: (3, 16)
┌──────────────┬────────┬──────┬───┬────────────┬─────────────┬──────────┐
│ name         ┆ height ┆ mass ┆ … ┆ birth_date ┆ screen_time ┆ bmi      │
│ ---          ┆ ---    ┆ ---  ┆   ┆ ---        ┆ ---         ┆ ---      │
│ str          ┆ u16    ┆ f64  ┆   ┆ date       ┆ duration[µs]┆ f64      │
```

```
┌──────────────┬────────┬──────┬─┬────────────┬──────────┬──────────┐
│ Han Solo     ┆ 180    ┆ 80.0 ┆ … ┆ 1948-06-01 ┆ 1h 12m 37s ┆ 24.691358 │
│ Leia Organa  ┆ 150    ┆ 49.0 ┆ … ┆ 1958-05-30 ┆ 1h 3m 40s  ┆ 21.777778 │
│ Luke Skywalker ┆ 172  ┆ 77.0 ┆ … ┆ 1958-05-30 ┆ 1h 58m 44s ┆ 26.027582 │
└──────────────┴────────┴──────┴─┴────────────┴──────────┴──────────┘
```

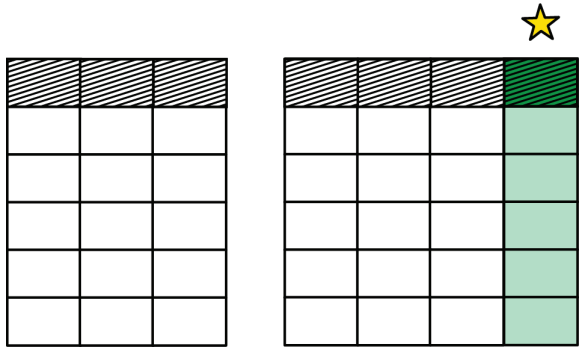


**Рис. 10.2** ❖ При добавлении столбца в датафрейм он занимает последнее место

Как видите, исходные столбцы сохранили свое место в датафрейме, а новый столбец был добавлен в конец.

## Столбцы можно перезаписывать

При создании в датафрейме нового столбца с именем, принадлежащим одному из существующих столбцов, этот столбец будет в результате перезаписан. В эту ловушку попадаются многие разработчики, так что будьте осторожны при использовании подобного кода:

```
df = pl.DataFrame({"a": [1, 2, 3]})
df.with_columns(pl.col("a") * 2)
```

Вывод:

```
shape: (3, 1)
┌─────┐
│ a   │
│ --- │
│ i64 │
╞═════╡
│ 2   │
│ 4   │
│ 6   │
└─────┘
```

В этом примере существующий столбец с именем `a` перезаписался с использованием новых значений.

При необходимости сохранить исходную колонку вы можете переименовать добавляемую, как показано ниже:

```
df.with_columns(a2=pl.col("a") * 2)
```

Вывод:

```
shape: (3, 2)
┌─────┬─────┐
│ a   ┆ a2  │
│ --- ┆ --- │
│ i64 ┆ i64 │
╞═════╪═════╡
│ 1   ┆ 2   │
│ 2   ┆ 4   │
│ 3   ┆ 6   │
└─────┴─────┘
```

Здесь мы назвали новый столбец `a2`, в результате чего исходный столбец сохранил свой первозданный вид.

При помощи `with_columns()` вы можете создать сразу несколько новых столбцов. Например, одновременно с вычислением индекса массы тела наших персонажей вы можете узнать, сколько полных лет им было в момент уничтожения главного реактора Звезды смерти[1]:

```
rebels.with_columns(
    bmi=pl.col("mass") / ((pl.col("height") / 100) ** 2),
    age_destroy=(
        (pl.date(1983, 5, 25) - pl.col("birth_date")).dt.total_days() / 365
    ).cast(pl.UInt8),
)
```

Вывод:

```
shape: (3, 17)
┌──────────────┬────────┬──────┬───┬──────────────┬──────────┬─────────────┐
│ name         ┆ height ┆ mass ┆ … ┆ screen_time  ┆ bmi      ┆ age_destroy │
│ ---          ┆ ---    ┆ ---  ┆   ┆ ---          ┆ ---      ┆ ---         │
│ str          ┆ u16    ┆ f64  ┆   ┆ duration[µs] ┆ f64      ┆ u8          │
╞══════════════╪════════╪══════╪═══╪══════════════╪══════════╪═════════════╡
│ Han Solo     ┆ 180    ┆ 80.0 ┆ … ┆ 1h 12m 37s   ┆ 24.691358 ┆ 35          │
│ Leia Organa  ┆ 150    ┆ 49.0 ┆ … ┆ 1h 3m 40s    ┆ 21.777778 ┆ 25          │
│ Luke Skywalker ┆ 172  ┆ 77.0 ┆ … ┆ 1h 58m 44s   ┆ 26.027582 ┆ 25          │
└──────────────┴────────┴──────┴───┴──────────────┴──────────┴─────────────┘
```

Выражения не могут зависеть друг от друга, поскольку они вычисляются параллельно. Это означает, что если вы хотите создать два новых столбца, второй не должен зависеть от первого. Если вам нужно именно это, вам придется разделить вызовы метода `with_columns()`.

---

[1] Второй Звезды смерти, конечно, из части «Звездные войны. Эпизод VI: Возвращение джедая».

Ниже показан пример с попыткой одновременно создать два столбца, где второй зависит от первого:

```
rebels.with_columns(
    bmi=pl.col("mass") / ((pl.col("height") / 100) ** 2),
    bmi_cat=pl.col("bmi").cut(
        [18.5, 25], labels=["Underweight", "Normal", "Overweight"]
    ),
)
```

Вывод:

```
ColumnNotFoundError: bmi

Resolved plan until failure:

        ---> FAILED HERE RESOLVING 'with_columns' <---
DF ["name", "height", "mass", "hair_color"]; PROJECT */15 COLUMNS
```

Решение состоит в повторном вызове метода `with_columns()`, как показано ниже:

```
(
    rebels.with_columns(
        bmi=pl.col("mass") / ((pl.col("height") / 100) ** 2)
    ).with_columns(
        bmi_cat=pl.col("bmi").cut(
            [18.5, 25], labels=["Underweight", "Normal", "Overweight"]
        )
    )
)
```

Вывод:

```
shape: (3, 17)
┌──────────────┬────────┬──────┬───┬──────────────┬──────────┬────────────┐
│ name         ┆ height ┆ mass ┆ … ┆ screen_time  ┆ bmi      ┆ bmi_cat    │
│ ---          ┆ ---    ┆ ---  ┆   ┆ ---          ┆ ---      ┆ ---        │
│ str          ┆ u16    ┆ f64  ┆   ┆ duration[µs] ┆ f64      ┆ cat        │
╞══════════════╪════════╪══════╪═══╪══════════════╪══════════╪════════════╡
│ Han Solo     ┆ 180    ┆ 80.0 ┆ … ┆ 1h 12m 37s   ┆ 24.691358 ┆ Normal     │
│ Leia Organa  ┆ 150    ┆ 49.0 ┆ … ┆ 1h 3m 40s    ┆ 21.777778 ┆ Normal     │
│ Luke Skywalker ┆ 172  ┆ 77.0 ┆ … ┆ 1h 58m 44s   ┆ 26.027582 ┆ Overweight │
└──────────────┴────────┴──────┴───┴──────────────┴──────────┴────────────┘
```

Для создания новых столбцов можно также воспользоваться методом датафрейма `select()`. Он позволяет выбрать подмножество существующих столбцов из датафрейма и одновременно дополнить его новыми.

Фактически вызов инструкции `df.with_columns(<новые столбцы>)` аналогичен вызову `df.select(pl.all(), <новые столбцы>)`, но является более прозрачным и понятным.

Помните, что в Python ключевые аргументы должны идти последними, так что если вам нужно, чтобы новые столбцы появились где-то в середине датафрейма, вам придется воспользоваться для их именования методом `Expr.alias()` вместо ключа.

Попытка вставить новый столбец в середину датафрейма с помощью ключевого аргумента приведет к показанной ниже ошибке:

```
starwars.select(
    "name",
    bmi=(pl.col("mass") / ((pl.col("height") / 100) ** 2)),
    "species",
)
```

Вывод:

```
SyntaxError: positional argument follows keyword argument (1147293163.py, line 5)
```

Давайте все сделаем правильно и посмотрим, какие персонажи обладают наибольшим индексом массы тела (а пока мы пишем код, можете попробовать угадать):

```
(
    starwars.select(
        "name",
        (pl.col("mass") / ((pl.col("height") / 100) ** 2)).alias("bmi"), ❶
        "species",
    )
    .drop_nulls()
    .top_k(5, by="bmi") ❷
)
```

❶ Обратите внимание, что сами столбцы `mass` и `height` можно не выбирать.
❷ Подробнее о методах `Expr.drop_nulls()` и `Expr.top_k()` мы поговорим в главе 11.

Вывод:

```
shape: (5, 3)
┌──────────────────────┬────────────┬─────────────────┐
│ name                 ┆ bmi        ┆ species         │
│ ---                  ┆ ---        ┆ ---             │
│ str                  ┆ f64        ┆ str             │
╞══════════════════════╪════════════╪═════════════════╡
│ Jabba Desilijic Tiure ┆ 443.428571 ┆ Hutt           │
│ Dud Bolt             ┆ 50.928022  ┆ Vulptereen      │
│ Yoda                 ┆ 39.02663   ┆ Yoda's species  │
│ Owen Lars            ┆ 37.874006  ┆ Human           │
│ IG-88                ┆ 35.0       ┆ Droid           │
└──────────────────────┴────────────┴─────────────────┘
```

**Создание новых столбцов**

По сути, метод `df.with_columns()` представляет собой синтаксический сахар для вызова метода `df.select()`, который выбирает все существующие столбцы в датафрейме и добавляет выражения, переданные в метод `df.with_columns()`. Для иллюстрации можете взглянуть на следующий пример:

```
df.with_columns(pl.lit(1).alias("ones"))
```

Вывод:

```
shape: (3, 2)
┌─────┬──────┐
│ a   ┆ ones │
│ --- ┆ ---  │
│ i64 ┆ i32  │
╞═════╪══════╡
│ 1   ┆ 1    │
│ 2   ┆ 1    │
│ 3   ┆ 1    │
└─────┴──────┘
```

Он аналогичен следующему вызову:

```
df.select(pl.all(), pl.lit(1).alias("ones"))
```

Вывод:

```
shape: (3, 2)
┌─────┬──────┐
│ a   ┆ ones │
│ --- ┆ ---  │
│ i64 ┆ i32  │
╞═════╪══════╡
│ 1   ┆ 1    │
│ 2   ┆ 1    │
│ 3   ┆ 1    │
└─────┴──────┘
```

# Операции для работы со столбцами

Помимо выбора и создания столбцов, существуют и другие разновидности операций, которые часто применяются в процессе анализа данных.

## Удаление столбцов

Иногда бывает удобнее указать, какие столбцы вы не хотите сохранить в результирующем датафрейме, вместо того чтобы перечислять все выбираемые столбцы. В этом случае вам лучше воспользоваться методом датафрейма *`drop()`*, как показано ниже:

```
rebels.drop("name", "films", "screen_time", strict=False) ❶
```

❶ Передача аргументу `strict` значения `False` позволяет указывать имена столбцов, отсутствующих в датафрейме, без боязни получить ошибку.

Вывод:

```
shape: (3, 13)
┌────────┬──────┬────────────┬───┬──────────────┬──────────────┬────────────┐
│ height ┆ mass ┆ hair_color ┆ … ┆ vehicles     ┆ starships    ┆ birth_date │
│ ---    ┆ ---  ┆ ---        ┆   ┆ ---          ┆ ---          ┆ ---        │
│ u16    ┆ f64  ┆ str        ┆   ┆ list[str]    ┆ list[str]    ┆ date       │
╞════════╪══════╪════════════╪═══╪══════════════╪══════════════╪════════════╡
│ 180    ┆ 80.0 ┆ brown      ┆ … ┆ null         ┆ ["Millennium ┆ 1948-06-01 │
│        ┆      ┆            ┆   ┆              ┆ Falcon"…     ┆            │
│ 150    ┆ 49.0 ┆ brown      ┆ … ┆ ["Imperial   ┆ null         ┆ 1958-05-30 │
│        ┆      ┆            ┆   ┆ Speeder B…   ┆              ┆            │
│ 172    ┆ 77.0 ┆ blond      ┆ … ┆ ["Snowspeeder" ┆ ["X-wing",   ┆ 1958-05-30 │
│        ┆      ┆            ┆   ┆ , "Imp…      ┆ "Imperial…   ┆            │
└────────┴──────┴────────────┴───┴──────────────┴──────────────┴────────────┘
```

Такого же результата можно добиться и с помощью метода `select()`, как показано ниже:

```
rebels.select(~cs.by_name("name", "films", "screen_time"))
```

или

```
rebels.select(cs.exclude("name", "films", "screen_time"))
```

## Переименование столбцов

Чисто технически вы можете задать новые имена существующим столбцам при помощи метода `select()`, но это бывает не слишком удобно, поскольку требует указания всех столбцов, которые нужно переименовать и сохранить в наборе. Гораздо удобнее бывает воспользоваться методом датафрейма *`rename()`*, принимающим на вход словарь или функцию. Продемонстрируем это на примере:

```
(
    rebels.rename({"homeworld": "planet", "mass": "weight"})
    .rename(lambda s: s.removesuffix("_color"))
    .select("name", "planet", "weight", "hair", "skin", "eye") ❶
)
```

❶ Метод `select()` здесь применяется только для того, чтобы имена новых столбцов выводились рядом.

Вывод:

```
shape: (3, 6)
┌──────────────┬──────────┬────────┬────────┬──────┬─────┐
│ name         ┆ planet   ┆ weight ┆ hair   ┆ skin ┆ eye │
```

```
| ---              | ---      | ---   | ---   | ---   | ---   |
| str              | str      | f64   | str   | str   | str   |
╞══════════════════╪══════════╪═══════╪═══════╪═══════╪═══════╡
| Han Solo         | Corellia | 80.0  | brown | fair  | brown |
| Leia Organa      | Alderaan | 49.0  | brown | light | brown |
| Luke Skywalker   | Tatooine | 77.0  | blond | fair  | blue  |
└──────────────────┴──────────┴───────┴───────┴───────┴───────┘
```

## Компоновка столбцов

Если у вас есть второй датафрейм или один или несколько объектов Series, имеющих такую же длину, как первый датафрейм, вы можете скомпоновать их по горизонтали с помощью метода *hstack()*. Давайте создадим два небольших датафрейма и один объект Series с цитатами, а затем объединим их вместе:

```
rebel_names = rebels.select("name")
rebel_colors = rebels.select(cs.ends_with("_color"))
rebel_quotes = pl.Series(
    "quote",
    [
        "You know, sometimes I amaze myself.",
        "That doesn't sound too hard.",
        "I have a bad feeling about this.",
    ],
)

(rebel_names.hstack(rebel_colors).hstack([rebel_quotes])) ❶
```

❶ Обратите внимание, что мы передаем список объектов Series.

Вывод:

```
shape: (3, 5)
┌──────────────────┬────────────┬────────────┬───────────┬──────────────────────┐
| name             | hair_color | skin_color | eye_color | quote                |
| ---              | ---        | ---        | ---       | ---                  |
| str              | str        | str        | str       | str                  |
╞══════════════════╪════════════╪════════════╪═══════════╪══════════════════════╡
| Han Solo         | brown      | fair       | brown     | You know, sometimes I |
|                  |            |            |           | amaze myself.        |
| Leia Organa      | brown      | light      | brown     | That doesn't sound   |
|                  |            |            |           | too hard.            |
| Luke Skywalker   | blond      | fair       | blue      | I have a bad feeling |
|                  |            |            |           | about this.          |
└──────────────────┴────────────┴────────────┴───────────┴──────────────────────┘
```

При подобном объединении датафреймов или объектов Series вам необходимо убедиться, что значения в них располагаются нужным вам образом, иначе вы рискуете получить непредсказуемый результат. В главе 14 мы подробнее разберем операции компоновки и объединения.

## Добавление индексов строк

Если вам необходимо добавить в датафрейм отдельный столбец с увеличивающимися значениями, вы можете воспользоваться удобным методом датафрейма *`with_row_index()`*. Этот метод принимает два опциональных аргумента `name` и `offset` со значениями по умолчанию `index` и `0` соответственно:

```
rebels.with_row_index(name="rebel_id", offset=1)
```

Вывод:

```
shape: (3, 16)
```

| rebel_id | name | height | … | birth_date | screen_time |
|---|---|---|---|---|---|
| u32 | str | u16 | | date | duration[μs] |
| 1 | Han Solo | 180 | … | 1948-06-01 | 1h 12m 37s |
| 2 | Leia Organa | 150 | … | 1958-05-30 | 1h 3m 40s |
| 3 | Luke Skywalker | 172 | … | 1958-05-30 | 1h 58m 44s |

## Заключение

В этой главе мы познакомились со способами выбора и создания столбцов, а также с операциями для работы со столбцами в датафрейме.

Вы узнали, что:

- выбирать столбцы из датафрейма можно разными способами;
- в большинстве случаев для указания нужных столбцов достаточно будет функции `pl.col()`;
- селекторы позволяют указывать столбцы на основе имен, типов данных и позиции в датафрейме;
- выбор столбцов по позиции может быть не лучшей идеей;
- селекторы предоставляют большую гибкость, поскольку они могут объединяться при помощи операторов для работы со множествами;
- новые столбцы в датафрейме можно создать как с помощью метода `select()`, так и посредством метода `with_columns()`;
- при создании нескольких столбцов, зависящих друг от друга, необходимо использовать множественные вызовы метода `with_columns()`;
- существует множество операций для работы со столбцами, включая их удаление, переименование и компоновку.

В следующей главе книги мы поговорим о фильтрации и сортировке строк в датафрейме.

# Глава 11

# Фильтрация и сортировка строк

Если в предыдущей главе мы в основном говорили о столбцах датафрейма, то эта глава будет целиком посвящена строкам[1]. Главным образом мы будем рассматривать две ключевые операции, связанные с работой над строками:

- фильтрация строк при помощи метода датафрейма `filter()`;
- сортировка строк при помощи метода датафрейма `sort()`.

Фильтрация позволяет выбрать подмножество строк на основе заданного фильтра. Сортировка делает возможным переопределение порядка следования строк в выборке в соответствии с вашими требованиями. Количество строк при этом остается неизменным. Помимо этих двух методов, мы также поговорим о сопутствующих операциях, связанных с фильтрацией и сортировкой.

На протяжении этой главы мы будем работать с небольшим датафреймом, содержащим информацию об электроинструменте, который вы можете встретить в гараже любого слесаря. Информация включает в себя тип инструмента, его код, название бренда, цену, индикатор возможности работать без подключения к сети и обороты в минуту (RPM). Датафрейм выглядит так:

```
tools = pl.read_csv("data/tools.csv")
tools
```

Вывод:

```
shape: (10, 6)
┌──────────────────┬─────────┬────────┬──────────┬───────┬──────┐
│ tool             ┆ product ┆ brand  ┆ cordless ┆ price ┆ rpm  │
│ ---              ┆ ---     ┆ ---    ┆ ---      ┆ ---   ┆ ---  │
│ str              ┆ str     ┆ str    ┆ bool     ┆ i64   ┆ i64  │
╞══════════════════╪═════════╪════════╪══════════╪═══════╪══════╡
│ Rotary Hammer    ┆ HR2230  ┆ Makita ┆ false    ┆ 199   ┆ 1050 │
```

[1] Методы, описанные в этой главе, могут также быть применены к ленивым датафреймам.

```
| Miter Saw             | GCM 8 SJL     | Bosch  | false | 391 | 5500  |
| Plunge Cut Saw        | DSP600ZJ      | Makita | true  | 459 | 6300  |
| Impact Driver         | DTD157Z       | Makita | true  | 156 | 3000  |
| Jigsaw                | PST 900 PEL   | Bosch  | false | 79  | 3100  |
| Angle Grinder         | DGA504ZJ      | Makita | true  | 229 | 8500  |
| Nail Gun              | DPSB2IN1-XJ   | DeWalt | true  | 129 | null  |
| Router                | POF 1400 ACE  | Bosch  | false | 185 | 28000 |
| Random Orbital Sander | DBO180ZJ      | Makita | true  | 199 | 11000 |
| Table Saw             | DWE7485       | DeWalt | false | 516 | 5800  |
└───────────────────────┴───────────────┴────────┴───────┴─────┴───────┘
```

Все инструкции по загрузке нужных файлов и установке пакетов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

Итак, приступим к фильтрации!

# Фильтрация строк

Строки в датафрейме Polars можно отфильтровать при помощи метода *`filter()`*. Применяя *фильтрацию*, мы можем ответить на вопросы о том, что чему равно и что больше чего. К примеру, мы можем выбрать все инструменты производителя Makita. Или все инструменты, способные работать без подключения к сети. На рис. 11.1 операция фильтрации показана схематически.

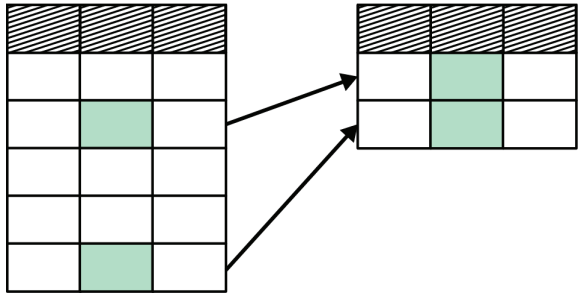


**Рис. 11.1** ❖ При фильтрации в наборе данных остаются строки, значения в которых в одном или нескольких столбцах удовлетворяют заданным критериям

Для фильтрации строк вам необходимо указать, какие именно строки вы хотите оставить. Это можно сделать с помощью выражений, имен столбцов и ограничений. Сначала рассмотрим первый вариант, поскольку он предоставляет наибольшую гибкость.

# Фильтрация на основе выражений

Первый способ отфильтровать данные состоит в использовании выражений. Вы уже встречались с подобными конструкциями в главе 7. Гибкость этого способа обусловлена возможностью использовать все типы операций сравнения и комбинировать их в любом порядке при помощи операторов булевой алгебры, таких как И и ИЛИ. Операции сравнения и операторы булевой алгебры мы подробно обсуждали в главе 9.

Выражения позволяют реализовать все мыслимые и немыслимые варианты фильтрации данных. Достаточно лишь убедиться, что используемые выражения вычисляются в булевы объекты Series. При этом значение `True` будет означать сохранение соответствующей строки в наборе данных, а `False` – ее исключение.

Давайте уже, наконец, приступим и для начала взглянем на инструменты производства Makita, которые могут работать без сети:

```
tools.filter(pl.col("cordless") & (pl.col("brand") == "Makita")) ❶
```

❶ Вам нет необходимости писать `pl.col("cordless") == True`, поскольку столбец `cordless` уже содержит булевы значения.

Вывод:

```
shape: (4, 6)
┌──────────────────────┬─────────┬────────┬──────────┬───────┬───────┐
│ tool                 ┆ product ┆ brand  ┆ cordless ┆ price ┆ rpm   │
│ ---                  ┆ ---     ┆ ---    ┆ ---      ┆ ---   ┆ ---   │
│ str                  ┆ str     ┆ str    ┆ bool     ┆ i64   ┆ i64   │
╞══════════════════════╪═════════╪════════╪══════════╪═══════╪═══════╡
│ Plunge Cut Saw       ┆ DSP600ZJ┆ Makita ┆ true     ┆ 459   ┆ 6300  │
│ Impact Driver        ┆ DTD157Z ┆ Makita ┆ true     ┆ 156   ┆ 3000  │
│ Angle Grinder        ┆ DGA504ZJ┆ Makita ┆ true     ┆ 229   ┆ 8500  │
│ Random Orbital Sander┆ DBO180ZJ┆ Makita ┆ true     ┆ 199   ┆ 11000 │
└──────────────────────┴─────────┴────────┴──────────┴───────┴───────┘
```

**Запятые вместо амперсандов**

Если ваше выражение состоит из нескольких частей, которые должны быть объединены при помощи логического И (`&`), вы можете передавать эти условия в метод `filter()` как отдельные аргументы. Это означает, что предыдущий код можно переписать так:

```
tools.filter(pl.col("cordless"), pl.col("brand") == "Makita")
```

Это может повысить читаемость вашего кода, если у вас аллергия на амперсанды. Помните, что этот подход не работает для операций логического ИЛИ (`|`).

# Фильтрация на основе имен столбцов

Второй способ фильтрации строк в датафрейме состоит в указании имен столбцов. Если столбец представлен булевым типом данных, как в случае со столбцом `cordless` в нашем датафрейме `tools`, вы можете использовать его имя в фильтре напрямую, без преобразования в выражение. К примеру, чтобы выбрать все беспроводные инструменты из нашего датафрейма, вы можете воспользоваться следующей конструкцией:

```
tools.filter("cordless")
```

Вывод:

```
shape: (5, 6)
┌──────────────────────┬──────────┬────────┬──────────┬───────┬───────┐
│ tool                 ┆ product  ┆ brand  ┆ cordless ┆ price ┆ rpm   │
│ ---                  ┆ ---      ┆ ---    ┆ ---      ┆ ---   ┆ ---   │
│ str                  ┆ str      ┆ str    ┆ bool     ┆ i64   ┆ i64   │
╞══════════════════════╪══════════╪════════╪══════════╪═══════╪═══════╡
│ Plunge Cut Saw       ┆ DSP600ZJ ┆ Makita ┆ true     ┆ 459   ┆ 6300  │
│ Impact Driver        ┆ DTD157Z  ┆ Makita ┆ true     ┆ 156   ┆ 3000  │
│ Angle Grinder        ┆ DGA504ZJ ┆ Makita ┆ true     ┆ 229   ┆ 8500  │
│ Nail Gun             ┆ DPSB2IN1-XJ ┆ DeWalt ┆ true ┆ 129   ┆ null  │
│ Random Orbital Sander ┆ DBO180ZJ ┆ Makita ┆ true     ┆ 199   ┆ 11000 │
└──────────────────────┴──────────┴────────┴──────────┴───────┴───────┘
```

Вы можете указать несколько имен столбцов, но помните, что все они должны иметь булев тип.



**Polars не поддерживает концепцию истинности и ложности, принятую в Python**

В языке программирования Python негласно принята концепция истинности и ложности, основанная на том, что значение можно считать истинным или ложным, если при приведении к булеву типу оно даст `True` или `False` соответственно. Таким образом, ложными значениями в Python можно считать сами значения `False`, а также нулевые значения и пустые последовательности, коллекции и строки. Все остальные значения неявным образом относятся к истинным. Следовательно, в Python условные конструкции (`my_name != ""`) и (`len(my_list) > 0`) можно переписать просто как `my_name` и `my_list` соответственно.

Исходя из этой концепции, привычной для Python, вы могли бы подумать, что в Polars тоже можно использовать не булевы столбцы и выражения при фильтрации. Однако это не так. Библиотека Polars написана на Rust и славится большей строгостью в сравнении с Python, чем объясняется допустимость использования в операциях фильтрации только булевых объектов Series.

Вы всегда можете преобразовать объект Series в булев при помощи операций сравнения. К примеру, для проверки на пустоту строки и заполненность списка вы можете воспользоваться конструкциями `pl.col("my_name") != ""` и `pl.col("my_list").list.len() > 0` соответственно.

# Фильтрация на основе ограничений

Третий способ использования метода `filter()` состоит в указании ограничений. *Ограничения* (constraint) состоят из имени столбца и значения[1]. Давайте снова оставим в датафрейме только беспроводные инструменты фирмы Makita:

```
tools.filter(cordless=True, brand="Makita")
```

Вывод:

```
shape: (4, 6)
```

| tool | product | brand | cordless | price | rpm |
|---|---|---|---|---|---|
| str | str | str | bool | i64 | i64 |
| Plunge Cut Saw | DSP600ZJ | Makita | true | 459 | 6300 |
| Impact Driver | DTD157Z | Makita | true | 156 | 3000 |
| Angle Grinder | DGA504ZJ | Makita | true | 229 | 8500 |
| Random Orbital Sander | DBO180ZJ | Makita | true | 199 | 11000 |

Как видите, имена столбцов в данном случае передаются методу `filter()` в качестве ключевых аргументов. Это может быть связано с некоторыми неудобствами следующего рода:

- имена передаваемых столбцов в этом случае могут состоять только из буквенных (a–z, A–Z) и цифровых (0–9) символов, а также символа подчеркивания (_), не могут начинаться с цифры и не могут совпадать с зарезервированными словами в Python, такими как `if`, `class`, `global` и т. д.;
- ограничения должны передаваться в метод `filter()` последними, если вы надумаете комбинировать их с двумя другими способами фильтрации (выражениями и именами столбцов);
- значения всегда должны быть указаны, включая значение `True`;
- поддерживается только операция равенства, и она должна указываться с помощью одного знака равенства, а не двух. Кроме того, согласно рекомендациям, принятым в языке Python, знаки равенства в этом случае не должны отделяться пробелами с двух сторон.

**Не стоит себя ограничивать**

Учитывая все приведенные выше неудобства при использовании ограничений, мы не рекомендуем применять их при выполнении фильтрации. Вместо этого мы отдаем предпочтение использованию выражений. Да, они более многословны, но при их использовании вы не будете ничем ограничены.

---

[1] Под капотом Polars преобразовывает переданные ограничения в конструкции вида `pl.col(<имя столбца>).eq(<значение>)`.

# Сортировка строк

При выполнении *сортировки* вы осуществляете переопределение порядка следования строк в выборке в соответствии с вашими требованиями. Количество строк в выборке и их содержимое при этом не меняются. При помощи сортировки можно, например, упорядочить строки по производителю, чтобы определить, инструменты какого бренда пользуются наибольшей популярностью. Также вы можете узнать, какой инструмент в нашем арсенале является самым дешевым. На рис. 11.2 операция сортировки показана схематически.

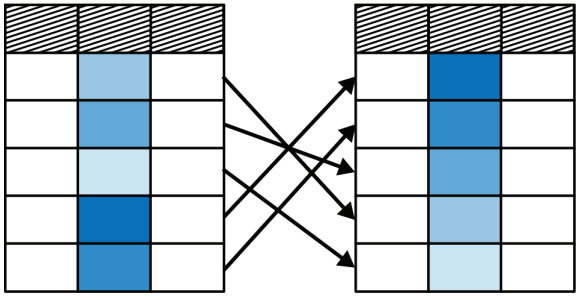


**Рис. 11.2** ❖ Сортировка строк осуществляется на основе значений в одном или нескольких столбцах

Наиболее часто вы будете прибегать к сортировке датафреймов по числовым столбцам, но ничто не мешает вам выполнять упорядочивание по строкам (в алфавитном порядке), датам или времени (в хронологическом порядке). Также вы можете сортировать данные по столбцам со сложными контейнерными типами данных, такими как `Struct` и `List`, о чем мы поговорим отдельно. В следующих нескольких разделах мы подробно разберем примеры сортировки по одному столбцу, нескольким столбцам и выражениям.

## Сортировка на основе одного столбца

Для упорядочивания данных вам необходимо воспользоваться методом датафрейма *`sort()`*. Наиболее простым вариантом его использования является передача имени столбца, как показано ниже:

```
tools.sort("price")
```

Вывод:

```
shape: (10, 6)
┌──────────────┬──────────────┬─────────┬──────────┬───────┬──────┐
│ tool         ┆ product      ┆ brand   ┆ cordless ┆ price ┆ rpm  │
```

```
| ---             | ---          | ---     | ---      | ---   | ---   |
| str             | str          | str     | bool     | i64   | i64   |
╞═════════════════╪══════════════╪═════════╪══════════╪═══════╪═══════╡
| Jigsaw          | PST 900 PEL  | Bosch   | false    | 79    | 3100  |
| Nail Gun        | DPSB2IN1-XJ  | DeWalt  | true     | 129   | null  |
| …               | …            | …       | …        | …     | …     |
| Plunge Cut Saw  | DSP600ZJ     | Makita  | true     | 459   | 6300  |
| Table Saw       | DWE7485      | DeWalt  | false    | 516   | 5800  |
└─────────────────┴──────────────┴─────────┴──────────┴───────┴───────┘
```

Как видите, по умолчанию значения в столбце сортируются в порядке возрастания. В табл. 11.1 приведены все принимаемые методом `sort()` аргументы и их описания.

***Таблица 11.1. Аргументы метода*** *`sort()`*

| Аргумент | Описание |
|---|---|
| `by` и `*more_by` | Столбец(цы) для сортировки. Принимае(ю)т выражения, включая селекторы. Строки воспринимаются как имена столбцов |
| `Descending` | Сортировка по убыванию значений. При сортировке по нескольким столбцам может быть указана для каждого столбца в виде последовательности булевых значений. Значение по умолчанию: `False` |
| `nulls_last` | Говорит о необходимости размещать пропущенные значения последними. Значение по умолчанию: `False` |
| `Multithreaded` | Сортировка с использованием нескольких потоков. Значение по умолчанию: `True`[1] |
| `maintain_order` | Указывает на необходимость сохранения исходного порядка при равенстве значений. Значение по умолчанию: `False` |

# Сортировка в обратном порядке

Вы можете изменить порядок сортировки, передав ключевому параметру `descending` метода `sort()` значение `True`, как показано ниже:

```
tools.sort("price", descending=True)
```

Вывод:

```
shape: (10, 6)
┌─────────────────┬──────────────┬─────────┬──────────┬───────┬───────┐
| tool            | product      | brand   | cordless | price | rpm   |
| ---             | ---          | ---     | ---      | ---   | ---   |
| str             | str          | str     | bool     | i64   | i64   |
╞═════════════════╪══════════════╪═════════╪══════════╪═══════╪═══════╡
| Table Saw       | DWE7485      | DeWalt  | false    | 516   | 5800  |
```

[1] Устанавливайте этот параметр в `False` только в случае, если ваш код на Polars является частью приложения, работающего в многопоточном режиме.

```
| Plunge Cut Saw | DSP600ZJ    | Makita | true     | 459   | 6300 |
| …              | …           | …      | …        | …     | …    |
| Nail Gun       | DPSB2IN1-XJ | DeWalt | true     | 129   | null |
| Jigsaw         | PST 900 PEL | Bosch  | false    | 79    | 3100 |
└────────────────┴─────────────┴────────┴──────────┴───────┴──────┘
```

**Вверх или вниз?**

Убедитесь, что вы используете именно аргумент `descending`, а не `ascending`, поскольку в противном случае вы получите ошибку, как показано ниже:

```
tools.sort("price", ascending=False)
```

Вывод:

```
TypeError: DataFrame.sort() got an unexpected keyword argument 'ascending'
```

Об этом легко забыть, если вы, например, часто используете в работе библиотеку pandas, где применяется аргумент `ascending=False` для упорядочивания значений по убыванию.

# Сортировка на основе нескольких столбцов

Для осуществления сортировки на основе нескольких столбцов вы можете передать их имена в виде ключевых аргументов, как показано в следующем примере:

```
tools.sort("brand", "price")
```

Вывод:

```
shape: (10, 6)
┌────────────────┬─────────────┬────────┬──────────┬───────┬──────┐
| tool           | product     | brand  | cordless | price | rpm  |
| ---            | ---         | ---    | ---      | ---   | ---  |
| str            | str         | str    | bool     | i64   | i64  |
╞════════════════╪═════════════╪════════╪══════════╪═══════╪══════╡
| Jigsaw         | PST 900 PEL | Bosch  | false    | 79    | 3100 |
| Router         | POF 1400 ACE| Bosch  | false    | 185   | 28000|
| …              | …           | …      | …        | …     | …    |
| Angle Grinder  | DGA504ZJ    | Makita | true     | 229   | 8500 |
| Plunge Cut Saw | DSP600ZJ    | Makita | true     | 459   | 6300 |
└────────────────┴─────────────┴────────┴──────────┴───────┴──────┘
```

По умолчанию применяется вариант сортировки по возрастанию для всех указанных столбцов. Если передать аргументу `descending` значение `True`, все указанные столбцы будут упорядочены по убыванию. Если вы хотите задать свой тип сортировки для каждого столбца, вы можете передать аргументу `descending` список булевых значений:

```
tools.sort("brand", "price", descending=[False, True])
```

Вывод:

```
shape: (10, 6)
```

| tool | product | brand | cordless | price | rpm |
|---|---|---|---|---|---|
| str | str | str | bool | i64 | i64 |
| Miter Saw | GCM 8 SJL | Bosch | false | 391 | 5500 |
| Router | POF 1400 ACE | Bosch | false | 185 | 28000 |
| … | … | … | … | … | … |
| Random Orbital Sander | DBO180ZJ | Makita | true | 199 | 11000 |
| Impact Driver | DTD157Z | Makita | true | 156 | 3000 |

Убедитесь, что количество переданных в списке значений соответствует количеству сортируемых столбцов.

## Сортировка на основе выражений

Метод `sort()` также может принимать одно или несколько выражений:

```
tools.sort(pl.col("rpm") / pl.col("price"))
```

Вывод:

```
shape: (10, 6)
```

| tool | product | brand | cordless | price | rpm |
|---|---|---|---|---|---|
| str | str | str | bool | i64 | i64 |
| Nail Gun | DPSB2IN1-XJ | DeWalt | true | 129 | null |
| Rotary Hammer | HR2230 | Makita | false | 199 | 1050 |
| … | … | … | … | … | … |
| Random Orbital Sander | DBO180ZJ | Makita | true | 199 | 11000 |
| Router | POF 1400 ACE | Bosch | false | 185 | 28000 |

Для сортировки датафрейма в этом случае создается временный объект Series, который не отображается в итоговом датафрейме.

Как и в случае с фильтрацией, использование выражений обеспечивает наибольшую гибкость. Но по опыту можно сказать, что гораздо чаще приходится сортировать датафрейм по присутствующим в нем столбцам.

# Сортировка вложенных типов данных

Также вы можете сортировать датафреймы в Polars по столбцам, характеризующимся вложенными типами данных, такими как структуры, списки и массивы (подробнее о них мы поговорим в главе 12). В Polars используется кодирование по строкам при сортировке данных. Списки и массивы упорядочиваются поэлементно. Это означает, что первые элементы списков сравниваются первыми, затем – вторые и т. д.:

```
lists = pl.DataFrame({"lists": [[2, 2], [2, 1, 3], [1]]})
lists.sort("lists")
```

Вывод:

```
shape: (3, 1)
┌───────────┐
│ lists     │
│ ---       │
│ list[i64] │
╞═══════════╡
│ [1]       │
│ [2, 1, 3] │
│ [2, 2]    │
└───────────┘
```

Структуры сортируются по полям – это значит, что сначала данные упорядочиваются по первому полю, затем по второму и т. д.:

```
structs = pl.DataFrame(
    {
        "structs": [
            {"a": 1, "b": 2, "c": 3},
            {"a": 1, "b": 3, "c": 1},
            {"a": 1, "b": 1, "c": 2},
        ]
    }
)

structs.sort("structs")
```

Вывод:

```
shape: (3, 1)
┌───────────┐
│ structs   │
│ ---       │
│ struct[3] │
╞═══════════╡
│ {1,1,2}   │
│ {1,2,3}   │
│ {1,3,1}   │
└───────────┘
```

Также вы можете предварительно извлечь или создать значения для сортировки. С целью демонстрации давайте создадим новый датафрейм `tools_collection` с группировкой инструментов по брендам в список структур:

```
tools_collection = tools.group_by("brand").agg(collection=pl.struct(pl.all()))
tools_collection
```

Вывод:

```
shape: (3, 2)
┌────────┬──────────────────────────────────────────────────────────────┐
│ brand  ┆ collection                                                   │
│ ---    ┆ ---                                                          │
│ str    ┆ list[struct[6]]                                              │
╞════════╪══════════════════════════════════════════════════════════════╡
│ Makita ┆ [{"Rotary Hammer","HR2230","Makita",false,199,1050}, {"Plung… │
│ Bosch  ┆ [{"Miter Saw","GCM 8 SJL","Bosch",false,391,5500}, {"Jigsaw"… │
│ DeWalt ┆ [{"Nail Gun","DPSB2IN1-XJ","DeWalt",true,129,null}, {"Table … │
└────────┴──────────────────────────────────────────────────────────────┘
```

Затем можно отсортировать списки по длине, которая представляет собой целочисленные значения:

```
tools_collection.sort(pl.col("collection").list.len(), descending=True)
```

Вывод:

```
shape: (3, 2)
┌────────┬──────────────────────────────────────────────────────────────┐
│ brand  ┆ collection                                                   │
│ ---    ┆ ---                                                          │
│ str    ┆ list[struct[6]]                                              │
╞════════╪══════════════════════════════════════════════════════════════╡
│ Makita ┆ [{"Rotary Hammer","HR2230","Makita",false,199,1050}, {"Plung… │
│ Bosch  ┆ [{"Miter Saw","GCM 8 SJL","Bosch",false,391,5500}, {"Jigsaw"… │
│ DeWalt ┆ [{"Nail Gun","DPSB2IN1-XJ","DeWalt",true,129,null}, {"Table … │
└────────┴──────────────────────────────────────────────────────────────┘
```

Также можно, например, упорядочить данные по средней цене для каждого бренда, как показано ниже:

```
tools_collection.sort(
    pl.col("collection")
    .list.eval(pl.element().struct.field("price"))
    .list.mean()
)
```

Вывод:

```
shape: (3, 2)
┌────────┬──────────────────────────────────────────────────────────────┐
│ brand  ┆ collection                                                   │
│ ---    ┆ ---                                                          │
│ str    ┆ list[struct[6]]                                              │
```

```
└─────────┴──────────────────────────────────────────────────────────────┘
┌─────────┬──────────────────────────────────────────────────────────────┐
| Bosch   | [{"Miter Saw","GCM 8 SJL","Bosch",false,391,5500}, {"Jigsaw","PST… |
| Makita  | [{"Rotary Hammer","HR2230","Makita",false,199,1050}, {"Plunge Cut… |
| DeWalt  | [{"Nail Gun","DPSB2IN1-XJ","DeWalt",true,129,null}, {"Table Saw",… |
└─────────┴──────────────────────────────────────────────────────────────┘
```

**Сначала материализуем, затем сортируем**

Иногда, как, например, в предыдущем примере, приходится иметь дело с не самыми простыми сценариями, в результате чего могут появляться сомнения в правильности выполненной сортировки. В таких случаях бывает полезно сначала создать новый столбец с помощью метода `with_columns()` для проверки значений, с которыми будет выполняться работа:

```
tools_collection.with_columns(
    mean_price=pl.col("collection")
    .list.eval(pl.element().struct.field("price"))
    .list.mean()
).sort("mean_price")
```

Вывод:

```
shape: (3, 3)
┌─────────┬──────────────────────────────────────┬────────────┐
| brand   | collection                           | mean_price |
| ---     | ---                                  | ---        |
| str     | list[struct[6]]                      | f64        |
└─────────┴──────────────────────────────────────┴────────────┘
┌─────────┬──────────────────────────────────────┬────────────┐
| Bosch   | [{"Miter Saw","GCM 8 SJL","Bosch",f… | 218.333333 |
| Makita  | [{"Rotary Hammer","HR2230","Makita"… | 248.4      |
| DeWalt  | [{"Nail Gun","DPSB2IN1-XJ","DeWalt"… | 322.5      |
└─────────┴──────────────────────────────────────┴────────────┘
```

Как видим, мы отсортировали датафрейм по правильным значениям. После этого можно спокойно превратить метод `with_columns()` обратно в метод `sort()`.

# Операции по работе со строками

Помимо фильтрации и сортировки, есть и другие операции для работы со строками, которые вам необходимо уметь выполнять.

## Фильтрация пропущенных значений

Некоторые алгоритмы и методы машинного обучения не способны корректно обрабатывать пропуски в данных. Метод *`drop_nulls()`* позволяет сохранить в датафрейме только строки без пропущенных значений. При этом вы можете указать столбцы, в которых стоит искать пропуски. Пример:

```
tools.drop_nulls("rpm").height
```

Вывод:

```
9
```

По умолчанию учитываются все столбцы, и в этом случае инструкция фактически эквивалентна следующей:

```
tools.filter(pl.all_horizontal(pl.all().is_not_null())).height
```

Вывод:

```
9
```

# Срезы

Часто вам бывает необходимо оставить в датафрейме только определенные строки, основываясь на их позициях, а не на значениях. Такие подмножества строк называются *срезами* (slice), и для их извлечения вы можете воспользоваться следующими методами:

- методы *`head()`* и *`tail()`* позволяют извлечь первые или последние несколько строк соответственно. К примеру, чтобы взять первые пять строк из датафрейма, можно воспользоваться инструкцией `df.head(5)`;
- с помощью метода *`slice()`* можно извлечь диапазон строк. Например, чтобы оставить только строки с третьей по седьмую, можно сделать следующий вызов: `df.slice(2, 7)`;
- метод *`gather_every()`* служит для извлечения строк через определенные разрывы. Допустим, каждую вторую строку в датафрейме можно извлечь так: `df.gather_every(2)`.

Конечно, вы можете комбинировать эти методы для создания сложных срезов. Например, следующий вызов позволит получить первые три строки из набора данных, состоящего из каждой второй строки из исходного датафрейма:

```
tools.with_row_index().gather_every(2).head(3)
```

Вывод:

```
shape: (3, 7)
```

| index | tool | product | … | cordless | price | rpm |
|---|---|---|---|---|---|---|
| u32 | str | str | | bool | i64 | i64 |
| 0 | Rotary Hammer | HR2230 | … | false | 199 | 1050 |
| 2 | Plunge Cut Saw | DSP600ZJ | … | true | 459 | 6300 |
| 4 | Jigsaw | PST 900 PEL | … | false | 79 | 3100 |

Методом `with_row_index()` мы здесь воспользовались, чтобы наглядно показать, какие строки в датафрейме остались.

**Порядок превыше всего!**

Обратите внимание, что перечисленные выше методы зависят от порядка, в котором располагаются строки в датафрейме. В целом же расчет на то, что строки будут располагаться в датафрейме определенным образом, считается не лучшей практикой при работе в Polars, поскольку это предполагает, что вы знаете о данных что-то, чего не знает Polars. Соответственно, движок Polars не может использовать эту информацию с целью оптимизации, и в целом это идет вразрез с декларативной природой Polars, при которой вы определяете операции на основе условий, а не расположения строк в датафрейме. Кроме того, такой расчет может помешать Polars выполнять свои вычисления параллельно.

# Верхние и нижние

Методы *top_k()* и *bottom_k()* позволяют оставить в наборе данных *k* строк с наибольшими или наименьшими значениями в указанном столбце. К примеру, три самых дорогих инструмента можно получить так:

```
tools.top_k(3, by="price")
```

Вывод:

```
shape: (3, 6)
┌──────────────┬──────────┬─────────┬──────────┬───────┬──────┐
│ tool         ┆ product  ┆ brand   ┆ cordless ┆ price ┆ rpm  │
│ ---          ┆ ---      ┆ ---     ┆ ---      ┆ ---   ┆ ---  │
│ str          ┆ str      ┆ str     ┆ bool     ┆ i64   ┆ i64  │
╞══════════════╪══════════╪═════════╪══════════╪═══════╪══════╡
│ Table Saw    ┆ DWE7485  ┆ DeWalt  ┆ false    ┆ 516   ┆ 5800 │
│ Plunge Cut Saw ┆ DSP600ZJ ┆ Makita ┆ true     ┆ 459   ┆ 6300 │
│ Miter Saw    ┆ GCM 8 SJL ┆ Bosch  ┆ false    ┆ 391   ┆ 5500 │
└──────────────┴──────────┴─────────┴──────────┴───────┴──────┘
```

**Никаких гарантий!**

Обратите внимание, что возвращенные значения не обязательно должны быть отсортированы, – они могут следовать в любом порядке. Если вам необходимо упорядочить их, примените к полученным результатам метод `sort()`.

# Семплирование

Метод *sample()* позволяет извлечь из набора данных подмножество случайных строк, также называемое семплом. К примеру, следующая инструкция поможет оставить 20 % случайных строк из датафрейма:

```
tools.sample(fraction=0.2)
```

Вывод:

```
shape: (2, 6)
┌──────────────┬──────────────┬─────────┬──────────┬───────┬───────┐
│ tool         ┆ product      ┆ brand   ┆ cordless ┆ price ┆ rpm   │
│ ---          ┆ ---          ┆ ---     ┆ ---      ┆ ---   ┆ ---   │
│ str          ┆ str          ┆ str     ┆ bool     ┆ i64   ┆ i64   │
╞══════════════╪══════════════╪═════════╪══════════╪═══════╪═══════╡
│ Rotary Hammer┆ HR2230       ┆ Makita  ┆ false    ┆ 199   ┆ 1050  │
│ Router       ┆ POF 1400 ACE ┆ Bosch   ┆ false    ┆ 185   ┆ 28000 │
└──────────────┴──────────────┴─────────┴──────────┴───────┴───────┘
```

Заметьте, что метод `sample()` работает только с материализованными датафреймами, поскольку ему необходимо заранее знать количество строк в наборе данных.

## Полусоединения

Еще один способ осуществления фильтрации данных заключается в использовании *полусоединения* (semi-join) с другим датафреймом. Допустим, у вас есть вспомогательный датафрейм с именем `saws`, содержащий все виды пил. Тогда вы можете оставить в датафрейме `tools` только строки, соответствующие перечню пил, следующим образом:

```
saws = pl.DataFrame(
    {
        "tool": [
                "Table Saw",
                "Plunge Cut Saw",
                "Miter Saw",
                "Jigsaw",
                "Bandsaw",
                "Chainsaw",
                "Seesaw",
        ]
    }
)
tools.join(saws, how="semi", on="tool")
```

Вывод:

```
shape: (4, 6)
┌──────────────┬──────────┬─────────┬──────────┬───────┬───────┐
│ tool         ┆ product  ┆ brand   ┆ cordless ┆ price ┆ rpm   │
│ ---          ┆ ---      ┆ ---     ┆ ---      ┆ ---   ┆ ---   │
│ str          ┆ str      ┆ str     ┆ bool     ┆ i64   ┆ i64   │
╞══════════════╪══════════╪═════════╪══════════╪═══════╪═══════╡
│ Miter Saw    ┆ GCM 8 SJL┆ Bosch   ┆ false    ┆ 391   ┆ 5500  │
│ Plunge Cut Saw┆ DSP600ZJ┆ Makita  ┆ true     ┆ 459   ┆ 6300  │
│ Jigsaw       ┆ PST 900 PEL┆ Bosch  ┆ false    ┆ 79    ┆ 3100  │
│ Table Saw    ┆ DWE7485  ┆ DeWalt  ┆ false    ┆ 516   ┆ 5800  │
└──────────────┴──────────┴─────────┴──────────┴───────┴───────┘
```

Подробнее о различного рода соединениях мы будем говорить в главе 14.

# Заключение

В этой главе мы подробно рассмотрели операции фильтрации и сортировки датафреймов, а также затронули другие операции для работы со строками.

Вы узнали, что:

- фильтрация на основе выражений дает наибольшую гибкость;
- для фильтрации необходимо использовать булевы объекты Series;
- фильтрация на основе ограничений имеет ряд неудобств;
- выражения, имена столбцов и ограничения, разделенные запятыми, можно объединять вместе при помощи логического И (`&`);
- чаще всего сортировка выполняется по одному столбцу, но можно упорядочивать данные и сразу по нескольким столбцам;
- для выполнения сортировки в обратном порядке необходимо воспользоваться аргументом `descending=True`;
- для сортировки столбцов с вложенными типами данных сначала нужно создать или извлечь из них значения, по которым будет выполняться упорядочивание;
- существует немало операций для работы с данными, включая извлечение срезов и семплирование.

В следующей главе мы погрузимся в работу с данными особых типов, такими как строки, категории, дата и время.

# Глава 12

# Работа с текстовыми, временными и вложенными типами данных

В главе 4 мы перечислили все возможные типы данных, используемые в Polars для хранения в объектах Series. Некоторые из этих типов требуют отдельного рассмотрения из-за наличия особых методов или необычного использования.

Эти типы данных можно условно разделить на текстовые, временные и вложенные. К текстовым мы будем относить типы `String`, `Categorical` и `Enum`, к временным – `Date`, `Datetime`, `Time` и `Duration`, а к вложенным – `List`, `Array` и `Struct`.

Все эти типы данных, за исключением `Enum`, обладают собственным пространством имен. Пространства имен служат для объединения однородных методов под общей крышей. К примеру, в пространстве имен `Expr.str` хранятся все методы для работы с типом данных `String`, а в пространстве имен `Expr.dt` – методы для работы с датой и временем.

В этой главе вы научитесь:

- создавать объекты Series, наполненные значениями текстового, временного и вложенного типов данных;
- работать с текстом при помощи типа данных `String`;
- использовать типы данных `Categorical` и `Enum` для эффективной работы с текстовыми данными;
- обрабатывать данные, связанные с датой и временем, с помощью типов данных `Date` и `Datetime`;
- хранить последовательности значений и вложенные данные посредством типов `List`, `Array` и `Struct`.

Все инструкции по загрузке нужных файлов и установке пакетов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

# Тип данных String

Тип данных *String* используется в Polars для представления текста, содержащего последовательности числовых, буквенных и иных символов. Этот тип данных подразумевает использование различных операций, типичных для работы с текстом, таких как поиск по шаблону, разделение на части и преобразование в разные форматы. В связи с этим типу `String` было выделено собственное пространство имен в Polars, называющееся *Expr.str*.

Более подробно о том, как текстовые данные хранятся в памяти, мы поговорим в главе 18.

## Методы типа данных String

Методы для работы с типом данных `String` можно условно разделить на три категории: методы преобразования, описания и манипуляции.

### *Методы преобразования*

Методы, перечисленные в табл. 12.1, позволяют преобразовывать значения типа `String` в и из разных форматов и типов.

***Таблица 12.1. Методы преобразования строковых данных***

| Метод | Описание |
|---|---|
| *Expr.str.decode()* | Декодирует значения в соответствии с указанной кодировкой |
| *Expr.str.encode()* | Кодирует значения в соответствии с указанной кодировкой |
| *Expr.str.json_decode()* | Декодирует строковые значения в формате JSON |
| *Expr.str.json_path_match()* | Извлекает первое совпадение из JSON в соответствии с переданным выражением JSONPath |
| *Expr.str.strptime()* | Преобразует столбец типа `String` в столбец типа `Date`, `Datetime` или `Time` |
| *Expr.str.to_date()* | Преобразует столбец типа `String` в столбец типа `Date` |
| *Expr.str.to_datetime()* | Преобразует столбец типа `String` в столбец типа `Datetime` |
| *Expr.str.to_decimal()* | Преобразует столбец типа `String` в столбец типа `Decimal` |

***Таблица 12.1*** *(окончание)*

| Метод | Описание |
|---|---|
| `Expr.str.to_integer()` | Преобразует столбец типа `String` в столбец типа `Int64` с заданным основанием системы счисления[1] |
| `Expr.str.to_time()` | Преобразует столбец типа `String` в столбец типа `Time` |

## Методы описания и запросов

Методы, перечисленные в табл. 12.2, позволяют извлекать нужные атрибуты строковых значений и запрашивать определенные шаблоны.

***Таблица 12.2. Методы описания строковых данных***

| Метод | Описание |
|---|---|
| `Expr.str.contains()` | Проверяет, содержит ли строка подстроку, соответствующую определенному шаблону |
| `Expr.str.contains_any()` | Использует алгоритм Ахо–Корасик[1] для поиска совпадений |
| `Expr.str.count_matches()` | Подсчитывает все последовательные непересекающиеся совпадения с регулярным выражением |
| `Expr.str.ends_with()` | Проверяет, оканчивается ли строка на заданную подстроку |
| `Expr.str.find()` | Возвращает индекс первого совпадения со строкой, содержащей шаблон |
| `Expr.str.len_bytes()` | Возвращает длину строки в байтах |
| `Expr.str.len_chars()` | Возвращает длину строки в символах |
| `Expr.str.starts_with()` | Проверяет, начинается ли строка с заданной подстроки |

## Методы манипуляции

В табл. 12.3 приведены методы, позволяющие управлять значениями типа `String` в объектах Series.

---

[1] Система счисления – это символический метод записи чисел, представление чисел с помощью письменных знаков. При этом в каждой системе используется свой диапазон буквенно-числовых значений. В контексте преобразования строк основание системы счисления определяет тип интерпретации значений. Без указания этого аргумента преобразование может оказаться двусмысленным. Например, число 101 можно представить в разных системах счисления по-разному: в десятичной – 101, в двоичной – 5, в шестнадцатеричной – 272. По умолчанию используется десятичная система счисления.

[1] Алгоритм Ахо–Корасик позволяет осуществлять быстрый поиск множества слов в тексте путем организации их в виде дерева и пропуска блоков при отсутствии совпадений. Это позволяет проверить весь текст за один проход.

***Таблица 12.3. Методы манипуляции***

| Метод | Описание |
|---|---|
| `Expr.str.concat()` | Вертикально объединяет строковые значения в столбце в одно строковое значение |
| `Expr.str.escape_regex()` | Возвращает строковые значения с экранированными метасимволами регулярных выражений |
| `Expr.str.explode()` | Возвращает столбец с отдельными строками для каждого строкового символа |
| `Expr.str.extract()` | Извлекает целевые группы из переданных шаблонов |
| `Expr.str.extract_all()` | Извлекает все совпадения для заданного шаблона регулярных выражений |
| `Expr.str.extract_groups()` | Извлекает все группы для заданного шаблона регулярных выражений |
| `Expr.str.extract_many()` | Использует алгоритм Ахо–Корасик для поиска множественных совпадений |
| `Expr.str.head()` | Возвращает первые n символов каждой строки в строковом объекте Series |
| `Expr.str.join()` | Вертикально объединяет строковые значения в столбце в одно строковое значение |
| `Expr.str.pad_end()` | Забивает остаток строкового значения до достижения нужной длины |
| `Expr.str.pad_start()` | Забивает начало строкового значения до достижения нужной длины |
| `Expr.str.replace()` | Заменяет первое вхождение подстроки на указанную строку |
| `Expr.str.replace_all()` | Заменяет все вхождения подстроки на указанную строку |
| `Expr.str.replace_many()` | Использует алгоритм Ахо–Корасик для множественной замены совпадений |
| `Expr.str.reverse()` | Возвращает строковое значение в обратном порядке |
| `Expr.str.slice()` | Извлекает подстроку из каждого строкового значения |
| `Expr.str.split()` | Разделяет строки на подстроки |
| `Expr.str.split_exact()` | Разделяет строки на подстроки с заданным количеством разделений |
| `Expr.str.splitn()` | Разделяет строки на подстроки с заданным максимальным количеством элементов |
| `Expr.str.strip_chars()` | Удаляет лидирующие и замыкающие символы |
| `Expr.str.strip_chars_end()` | Удаляет замыкающие символы |
| `Expr.str.strip_chars_start()` | Удаляет лидирующие символы |
| `Expr.str.strip_prefix()` | Удаляет префикс |
| `Expr.str.strip_suffix()` | Удаляет суффикс |
| `Expr.str.tail()` | Возвращает последние n символов каждой строки в строковом объекте Series |
| `Expr.str.to_lowercase()` | Приводит строки к нижнему регистру |
| `Expr.str.to_titlecase()` | Приводит строки к нижнему регистру с капитализацией начальных букв всех слов в предложении |
| `Expr.str.to_uppercase()` | Приводит строки к верхнему регистру |
| `Expr.str.zfill()` | Забивает начало строкового значения нулями до достижения нужной длины |

# Примеры работы со строками

Теперь давайте рассмотрим несколько полезных примеров работы со строковыми значениями. Но сначала создадим небольшой датафрейм следующего вида:

```
corpus = pl.DataFrame(
    {
        "raw_text": [
            "  Data Science is amazing ",
            "Data_analysis > Data entry",
            " Python&Polars; Fast",
        ]
    }
)

corpus
```

Вывод:

```
shape: (3, 1)
┌──────────────────────────┐
│ raw_text                 │
│ ---                      │
│ str                      │
╞══════════════════════════╡
│   Data Science is amazing  │
│ Data_analysis > Data entry │
│  Python&Polars; Fast       │
└──────────────────────────┘
```

На примере этого простого датафрейма мы продемонстрируем несколько методов для работы со строками в Polars. Начнем с очистки строк:

```
corpus = corpus.with_columns(
    processed_text=pl.col("raw_text") ❶
    .str.strip_chars() ❷
    .str.to_lowercase() ❸
    .str.replace_all("_", " ") ❹
)
corpus
```

❶ Создаем новый столбец с обработанным текстом и называем его соответствующим образом.

❷ Метод `Expr.str.strip_chars()` позволяет избавиться от паразитных символов в начале и конце строки. Поскольку мы не передали методу конкретный символ, будут удаляться пробелы.

❸ Приведение строк к нижнему регистру может облегчить дальнейшую работу с ними, исключая возможные ошибки в шаблонах, связанные с регистром.

❹ Вам может понадобиться замена всех символов подчеркивания пробелами при работе с именами файлов и ссылками.

Вывод:

```
shape: (3, 2)
┌──────────────────────────┬──────────────────────────┐
│ raw_text                 ┆ processed_text           │
│ ---                      ┆ ---                      │
│ str                      ┆ str                      │
╞══════════════════════════╪══════════════════════════╡
│   Data Science is amazing ┆ data science is amazing  │
│ Data_analysis > Data entry ┆ data analysis > data entry │
│  Python&Polars; Fast     ┆ python&polars; fast      │
└──────────────────────────┴──────────────────────────┘
```

Теперь, когда мы немного очистили входные данные, приступим непосредственно к манипуляции с ними. Одними из самых распространенных операций при работе с текстом являются операции извлечения срезов и разбиения строк:

```
corpus.with_columns(
    first_5_chars=pl.col("processed_text").str.slice(0, 5), ❶
    first_word=pl.col("processed_text")
    .str.split(" ") ❷
    .list.get(0), ❸
    second_word=pl.col("processed_text").str.split(" ").list.get(1), ❹
)
```

❶ Из столбца `processed_text` мы извлекаем пять первых символов и сохраняем их в столбце с говорящим именем `first_5_chars`.
❷ Разделяем строки в столбце `processed_text` по пробелам. В результате получим список строк длины, равной количеству пробелов плюс один. О списках мы поговорим далее в этой главе.
❸ Извлекаем первый элемент из списка, т. е. первое слово.
❹ Извлекаем второй элемент из списка, т. е. второе слово.

Вывод:

```
shape: (3, 5)
┌──────────────┬──────────────┬──────────────┬──────────────┬──────────────┐
│ raw_text     ┆ processed_tex ┆ first_5_chars ┆ first_word   ┆ second_word  │
│ ---          ┆ t            ┆ ---          ┆ ---          ┆ ---          │
│ str          ┆ ---          ┆ str          ┆ str          ┆ str          │
│              ┆ str          ┆              ┆              ┆              │
╞══════════════╪══════════════╪══════════════╪══════════════╪══════════════╡
│ Data Science ┆ data science ┆ data         ┆ data         ┆ science      │
│ is am…       ┆ is amaz…     ┆              ┆              ┆              │
│ Data_analysis ┆ data analysis ┆ data         ┆ data         ┆ analysis     │
│ > Data…      ┆ > data…      ┆              ┆              ┆              │
│ Python&Polars; ┆ python&polars ┆ pytho        ┆ python&polars ┆ fast         │
│ Fast         ┆ ; fast       ┆              ┆ ;            ┆              │
└──────────────┴──────────────┴──────────────┴──────────────┴──────────────┘
```

Вы также можете запросить у строковых значений нужную вам информацию, как показано ниже:

```
corpus.with_columns(
    len_chars=pl.col("processed_text").str.len_chars(), ❶
    len_bytes=pl.col("processed_text").str.len_bytes(), ❷
    count_a=pl.col("processed_text").str.count_matches("a"), ❸
)
```

❶ Вычисляем количество символов в строках.
❷ Вычисляем количество байтов, занимаемых строками в памяти.
❸ Определяем частоту встречаемости буквы *a* в строках.

Вывод:

```
shape: (3, 5)
```

| raw_text | processed_text | len_chars | len_bytes | count_a |
|---|---|---|---|---|
| str | str | u32 | u32 | u32 |
| Data Science is amazing | data science is amazing | 23 | 23 | 4 |
| Data_analysis > Data entry | data analysis > data entry | 26 | 26 | 6 |
| Python&Polars; Fast | python&polars; fast | 19 | 19 | 2 |

**Длина строк и быстродействие**

В Polars значения типа `String` хранятся в кодировке UTF-8, характеризующейся переменной длиной. Это означает, что для хранения в памяти разных символов может быть задействовано разное количество байтов. В зависимости от типа символа кодировка UTF-8 может использовать от одного до четырех байт для его хранения в памяти.

В предыдущем примере значения в столбцах `len_chars` и `len_bytes` оказались одинаковыми, поскольку мы использовали только символы из диапазона ASCII, включающего все буквы латинского алфавита, цифры и базовые символы. При использовании символов, отличных от стандартных, например азиатских иероглифов, кириллицы или эмодзи, может быть задействовано более одного байта на символ. В этом случае вам, возможно, стоит воспользоваться методом `Expr.str.len_chars()` вместо `Expr.str.len_bytes()`.

Помните, что метод `Expr.str.len_bytes()` обладает гораздо большим быстродействием в сравнении с методом `Expr.str.len_chars()`. На выполнение метода `Expr.str.len_bytes()` требуется постоянное время, не зависящее от длины строки. В то же время скорость выполнения метода `Expr.str.len_chars()` линейно зависит от объема данных. Таким образом, чем длиннее ваши строки, тем дольше вам придется ждать.

В следующем коде мы воспользуемся регулярными выражениями для поиска всех хештегов в строке:

```
posts = pl.DataFrame(
    {"post": ["Loving #python and #polars!", "A boomer post without a hashtag"]}
)

hashtag_regex = r"#(\w+)" ❶
```

```
posts.with_columns(
    hashtags=pl.col("post").str.extract_all(hashtag_regex) ❷
)
```

❶ Определяем шаблон регулярного выражения, соответствующий символу решетки с последующими буквенно-цифровыми символами. Здесь `\w` означает вхождение любого количества буквенно-цифровых символов. Под этими символами подразумеваются символы в диапазонах a–z, A–Z или 0–9, а также символ подчеркивания (_). Знак плюс (+) означает, что предшествующий шаблон должен встретиться один или более раз, что позволит извлечь целые слова, а не только первые буквы.
❷ Извлекаем все совпадения из столбца `post` и сохраняем их в столбце `hashtags`.

Вывод:

```
shape: (2, 2)
┌──────────────────────────────┬──────────────────────┐
│ post                         ┆ hashtags             │
│ ---                          ┆ ---                  │
│ str                          ┆ list[str]            │
╞══════════════════════════════╪══════════════════════╡
│ Loving #python and #polars!  ┆ ["#python", "#polars"] │
│ A boomer post without a hashtag ┆ []                │
└──────────────────────────────┴──────────────────────┘
```

# Тип данных Categorical

С помощью типа данных *`Categorical`* можно очень эффективно закодировать строковый объект Series. При использовании типа `String` все значения хранятся в физической памяти по отдельности, даже если они повторяются.

В свою очередь, тип данных `Categorical` использует так называемый *строковый кеш* (String Cache). Строковый кеш представляет собой скрытый под капотом словарь, в котором хранятся уникальные значения типа `String` и соответствующие им целочисленные (типа данных `UInt32`, если быть точными) представления для каждого уникального строкового значения в этом объекте Series. Таким образом, вместо хранения всех объемных значений типа `String` в объекте Series хранятся только их целочисленные представления. Эти числа называются *физическим представлением* (physical representation), а сами значения типа `String` – *лексическим представлением* (lexical representation).

При содержании в объекте Series большого количества повторяющихся значений типа `String` это позволяет значительно повысить эффективность хранения и ускорить выполнение операций – это связано с тем, что сравнение значений типа `String` является довольно дорогостоящим. Значения типа `Categorical` хранятся в двух частях: в виде словаря и индексов.

Давайте поближе познакомимся с типом данных `Categorical` и его методами. Для этого сначала создадим датафрейм со столбцом типа `Categorical`. Дополнительно мы добавим столбец с физическими представлениями хранящихся значений:

```
cats = pl.DataFrame(
    {"name": ["Persian cat", "Siamese Cat", "Lynx", "Lynx"]},
    schema={"name": pl.Categorical},
)

cats.with_columns(name_physical=pl.col("name").to_physical())
```

Вывод:

```
shape: (4, 2)
┌─────────────┬─────────────────┐
│ name        ┆ name_physical   │
│ ---         ┆ ---             │
│ cat         ┆ u32             │
╞═════════════╪═════════════════╡
│ Persian cat ┆ 0               │
│ Siamese Cat ┆ 1               │
│ Lynx        ┆ 2               │
│ Lynx        ┆ 2               │
└─────────────┴─────────────────┘
```

# Методы для работы с типом Categorical

Тип данных `Categorical` может похвастаться лишь одним методом, для которого мы выделили целую табл. 12.4.

***Таблица 12.4. Методы для работы с типом Categorical***

| Метод | Описание |
|---|---|
| *`Expr.cat.get_categories()`* | Извлекает категории, хранящиеся в типе данных `Categorical` |

# Примеры работы с типом Categorical

Порядок следования значений типа `Strings` в столбце определяет внешний вид значения типа `Categorical` и словаря. Даже если в другом датафрейме будет столбец с такими же уникальными значениями в исходном столбце, значение типа `Categorical` будет отличаться, если порядок следования строковых значений будет иным. Причина в том, что порядок следования элементов в словаре, а значит, и физических представлений, будет другим, что видно в показанном ниже примере:

```
more_cats = pl.DataFrame(
    {"name": ["Maine Coon Cat", "Lynx", "Lynx", "Siamese Cat"]},
    schema={"name": pl.Categorical},
)

more_cats.with_columns(name_physical=pl.col("name").to_physical())
```

Вывод:

```
shape: (4, 2)
┌──────────────┬─────────────────┐
│ name         ┆ name_physical   │
│ ---          ┆ ---             │
│ cat          ┆ u32             │
╞══════════════╪═════════════════╡
│ Maine Coon Cat ┆ 0             │
│ Lynx         ┆ 1               │
│ Lynx         ┆ 1               │
│ Siamese Cat  ┆ 2               │
└──────────────┴─────────────────┘
```

Вследствие этого при попытке объединить датафреймы по столбцу со значениями типа `Categorical` вы увидите предупреждение `CategoricalRemappingWarning`:

```
cats.join(more_cats, on="name")
```

Вывод:

```
CategoricalRemappingWarning: Local categoricals have different encodings, expensive re-
encoding is done to perform this merge operation. Consider using a String Cache or an Enum
type if the categories are known in advance
  cats.join(more_cats, on="name")
```

Вывод:

```
shape: (5, 1)
┌──────────────┐
│ name         │
│ ---          │
│ cat          │
╞══════════════╡
│ Lynx         │
│ Lynx         │
│ Lynx         │
│ Lynx         │
│ Siamese Cat  │
└──────────────┘
```

Для эффективного объединения двух объектов Series категориального типа вы можете собрать и значения в одном строковом кеше. Сделать это можно при помощи *глобального строкового кеша* (Global String Cache), который представляет собой кеш, разделенный между всеми объектами типа `Categorical`. Это позволяет хранить данные обо всех объектах категориального типа в одном месте, что предотвращает появление несоответствий. По умолчанию глобальный строковый кеш отключен, поскольку хранить все объекты типа `Categorical` в одном кеше бывает накладно с точки зрения быстродействия.

Связано это с необходимостью блокировать кеш на время доступа к нему, что приводит к необходимости ожидания доступа другими потоками.

В следующем фрагменте кода показано, как можно содержать несколько объектов типа `Categorical` в одном кеше при помощи контекстного менеджера *`pl.StringCache`*:

```
with pl.StringCache():
    left = pl.DataFrame(
        {
            "categorical_column": ["value3", "value2", "value1"],
            "other": ["a", "b", "c"],
        },
        schema={"categorical_column": pl.Categorical, "other": pl.String},
    )
    right = pl.DataFrame(
        {
            "categorical_column": ["value2", "value3", "value4"],
            "other": ["d", "e", "f"],
        },
        schema={"categorical_column": pl.Categorical, "other": pl.String},
    )
```

Теперь вы даже за пределами контекстного менеджера можете безопасно объединять два датафрейма, содержащих столбцы типа `Categorical`, как показано ниже:

```
left.join(right, on="categorical_column")
```

Вывод:

```
shape: (2, 3)
┌──────────────────┬───────┬──────────────┐
│ categorical_column ┆ other ┆ other_right │
│ ---              ┆ ---   ┆ ---          │
│ cat              ┆ str   ┆ str          │
╞══════════════════╪═══════╪══════════════╡
│ value2           ┆ b     ┆ d            │
│ value3           ┆ a     ┆ e            │
└──────────────────┴───────┴──────────────┘
```

Также вы можете активировать глобальный строковый кеш с помощью функции *`pl.enable_string_cache()`*:

```
pl.enable_string_cache()
```

Но учтите, что в этом случае глобальный строковый кеш будет задействован постоянно, что может быть неоптимально в сравнении с использованием контекстного менеджера.

Чтобы извлечь все уникальные категории, хранящиеся в столбце с типом `Categorical`, необходимо воспользоваться единственным методом из пространства имен *`cat`*, а именно `Expr.cat.get_categories()`:

```
right.select(pl.col("categorical_column").cat.get_categories())
```

Вывод:

```
shape: (3, 1)
┌──────────────────┐
│ categorical_column │
│ ---              │
│ str              │
╞══════════════════╡
│ value2           │
│ value3           │
│ value4           │
└──────────────────┘
```

Последнее, о чем стоит упомянуть применительно к столбцам с типом `Categorical`, – это о порядке, в котором располагаются значения в них при сортировке. Здесь есть два варианта:

- *физический* (по умолчанию): для упорядочивания используются физические (целочисленные) представления;
- *лексический*: для упорядочивания используются значения с типом `String`.

Эти опции можно установить при создании столбца типа `Categorical`. Также вы можете переключаться между вариантами представления данных.

Для начала создадим датафрейм:

```
sorting_comparison_df = cats.select(cat_lexical=pl.col("name")).with_columns(
    cat_physical=pl.col("cat_lexical").to_physical()
)

sorting_comparison_df
```

Вывод:

```
shape: (4, 2)
┌─────────────┬──────────────┐
│ cat_lexical ┆ cat_physical │
│ ---         ┆ ---          │
│ cat         ┆ u32          │
╞═════════════╪══════════════╡
│ Persian cat ┆ 0            │
│ Siamese Cat ┆ 1            │
│ Lynx        ┆ 2            │
│ Lynx        ┆ 2            │
└─────────────┴──────────────┘
```

Отсортируем столбец `cat_lexical` по физическому представлению. Результат сортировки виден по расположению значений в столбце `cat_physical`:

```
sorting_comparison_df.with_columns(
    pl.col("cat_lexical").cast(pl.Categorical("physical"))
).sort(by="cat_lexical")
```

Вывод:

```
shape: (4, 2)
┌─────────────┬──────────────┐
│ cat_lexical ┆ cat_physical │
│ ---         ┆ ---          │
│ cat         ┆ u32          │
╞═════════════╪══════════════╡
│ Persian cat ┆ 0            │
│ Siamese Cat ┆ 1            │
│ Lynx        ┆ 2            │
│ Lynx        ┆ 2            │
└─────────────┴──────────────┘
```

**Переходим к лексике**

По умолчанию значения в столбцах с типом `Categorical` сортируются по физическому представлению. Но вы всегда можете перейти к сортировке в лексическом порядке.

Ниже показан пример упорядочивания значений в столбце с типом `Categorical` по лексическому представлению, что видно по содержимому столбца `cat_lexical`:

```
sorting_comparison_df.with_columns(
    pl.col("cat_lexical").cast(pl.Categorical("lexical"))
).sort(by="cat_lexical")
```

Вывод:

```
shape: (4, 2)
┌─────────────┬──────────────┐
│ cat_lexical ┆ cat_physical │
│ ---         ┆ ---          │
│ cat         ┆ u32          │
╞═════════════╪══════════════╡
│ Lynx        ┆ 2            │
│ Lynx        ┆ 2            │
│ Persian cat ┆ 0            │
│ Siamese Cat ┆ 1            │
└─────────────┴──────────────┘
```

# Тип данных Enum

Если вы знаете заранее, какие значения могут встречаться в объекте Series, вы можете присвоить ему специальный тип данных *`Enum`*. В настоящее время этот тип внутренне используется типом данных `Categorical`, но в будущем, возможно, получит собственную реализацию. В следующем примере показано, как можно создать объект типа `Enum` с именем `bear_enum_dtype` и объект Series, его реализующий:

```
bear_enum_dtype = pl.Enum(["Polar", "Panda", "Brown"])

bear_enum_series = pl.Series(
    ["Polar", "Panda", "Brown", "Brown", "Polar"], dtype=bear_enum_dtype
)

bear_cat_series = pl.Series(
    ["Polar", "Panda", "Brown", "Brown", "Polar"], dtype=pl.Categorical
)
```

Тип данных `Enum` является довольно новым в Polars и на момент написания книги не обладает собственным пространством имен.

# Типы данных, связанные с датой и временем

Типы данных, связанные с датой и временем, представляют собой обособленные структуры, помогающие эффективно работать с календарными значениями в данных. Эти типы позволяют выполнять арифметические операции, операции сравнения и прочие применительно к значениям, представляющим временные метки.

В табл. 12.5 перечислены все четыре типа данных, предназначенных для хранения информации, связанной с датой и временем.

***Таблица 12.5. Типы данных, связанные с датой и временем***

| Тип данных | Описание | Пример | Хранение |
|---|---|---|---|
| `Date` | Представляет календарные даты без времени | Дни рождения | Количество дней, прошедших с начала эпохи Unix (1 января 1970 года), выраженное в типе данных `Int32` |
| `Datetime` | Представляет календарные даты со временем | Даты и время в логах | Количество временных отрезков, прошедших с начала эпохи Unix (1 января 1970 года), выраженное в типе данных `Int64` и имеющее разный уровень детализации |
| `Duration` | Представляет временные интервалы между двумя точками во времени. Аналогичен типу данных `timedelta` в Python | Время, прошедшее между двумя событиями | Значения с типом данных `Int64`, получаемые в результате вычитания объектов с типами `Date/ Datetime` |
| `Time` | Представляет только время без дат | Расписание дня | Количество наносекунд, прошедших с полуночи, выраженное в типе данных `Int64` |

# Методы для работы с календарными типами данных

Пространство имен *Expr.dt* содержит большое количество методов для преобразования, описания и манипуляции с календарными типами данных.

## Методы преобразования

Методы, показанные в табл. 12.6, предназначены для преобразования календарных данных из и в другие форматы и типы.

***Таблица 12.6. Методы преобразования календарных типов данных***

| Метод | Описание |
|---|---|
| *Expr.dt.cast_time_unit()* | Приводит значения к другим единицам измерения времени |
| *Expr.dt.strftime()* | Приводит столбцы с типами `Date`/`Time`/`Datetime` к типу `String` в соответствии с заданным форматом |
| *Expr.dt.to_string()* | Приводит столбцы с типами `Date`/`Time`/`Datetime` к типу `String` в соответствии с заданным форматом |

## Методы описания и запросов

Методы, перечисленные в табл. 12.7, предназначены для извлечения атрибутов календарных данных.

***Таблица 12.7. Методы описания календарных данных***

| Метод | Описание |
|---|---|
| *Expr.dt.base_utc_offset()* | Извлекает базовое смещение от стандарта UTC |
| *Expr.dt.century()* | Извлекает век из представления даты |
| *Expr.dt.date()* | Извлекает дату из типа `Datetime` |
| *Expr.dt.datetime()* | Возвращает данные в формате `Datetime` |
| *Expr.dt.day()* | Извлекает день из представления даты |
| *Expr.dt.dst_offset()* | Извлекает текущее смещение, обычно примененное вследствие перехода на летнее/зимнее время |
| *Expr.dt.epoch()* | Возвращает время, прошедшее с начала эпохи Unix в заданных единицах измерения |
| *Expr.dt.hour()* | Извлекает часы из представления даты |
| *Expr.dt.is_leap_year()* | Возвращает индикатор того, является ли год високосным |
| *Expr.dt.iso_year()* | Извлекает год ISO из представления даты |
| *Expr.dt.microsecond()* | Извлекает количество микросекунд из представления даты |
| *Expr.dt.millennium()* | Извлекает тысячелетие из представления даты |
| *Expr.dt.millisecond()* | Извлекает количество миллисекунд из представления даты |
| *Expr.dt.minute()* | Извлекает количество минут из представления даты |
| *Expr.dt.month()* | Извлекает месяц из представления даты |
| *Expr.dt.nanosecond()* | Извлекает количество наносекунд из представления даты |
| *Expr.dt.ordinal_day()* | Извлекает порядковый номер дня из представления даты |
| *Expr.dt.quarter()* | Извлекает номер квартала из представления даты |

***Таблица 12.7*** *(окончание)*

| Метод | Описание |
|---|---|
| *Expr.dt.second()* | Извлекает количество секунд из представления даты |
| *Expr.dt.time()* | Извлекает время из представления даты |
| *Expr.dt.timestamp()* | Возвращает метку времени в заданных единицах измерения |
| *Expr.dt.total_days()* | Извлекает количество полных дней из значения типа `Duration` |
| *Expr.dt.total_hours()* | Извлекает количество полных часов из значения типа `Duration` |
| *Expr.dt.total_microseconds()* | Извлекает количество микросекунд из значения типа `Duration` |
| *Expr.dt.total_milliseconds()* | Извлекает количество миллисекунд из значения типа `Duration` |
| *Expr.dt.total_minutes()* | Извлекает количество полных минут из значения типа `Duration` |
| *Expr.dt.total_nanoseconds()* | Извлекает количество наносекунд из значения типа `Duration` |
| *Expr.dt.total_seconds()* | Извлекает количество полных секунд из значения типа `Duration` |
| *Expr.dt.year()* | Извлекает номер года из представления даты |

## *Методы для манипуляции календарными данными*

Методы, перечисленные в табл. 12.8, позволяют выполнять нужные манипуляции применительно к календарным данным.

***Таблица 12.8. Методы для манипуляции календарными данными***

| Метод | Описание |
|---|---|
| *Expr.dt.add_business_days()* | Добавляет смещение на указанное количество рабочих дней |
| *Expr.dt.combine()* | Пытается наивным образом собрать значение типа `Datetime` на основе существующих выражений типа `Date`/`Datetime` и `Time` |
| *Expr.dt.convert_time_zone()* | Преобразует часовой пояс для выражения типа `Datetime` |
| *Expr.dt.month_start()* | Выполняет откат на первый день месяца |
| *Expr.dt.month_end()* | Выполняет откат на последний день месяца |
| *Expr.dt.offset_by()* | Осуществляет заданное смещение временной метки |
| *Expr.dt.replace_time_zone()* | Заменяет часовой пояс для выражения типа `Datetime` |
| *Expr.dt.round()* | Делит диапазон `Date`/`Datetime` на отрезки |
| *Expr.dt.truncate()* | Делит диапазон `Date`/`Datetime` на отрезки |
| *Expr.dt.week()* | Извлекает номер недели из представления даты |
| *Expr.dt.weekday()* | Извлекает день недели из представления даты |
| *Expr.dt.with_time_unit()* | Устанавливает единицы измерения для выражения типа `Datetime` или `Duration` |

# Примеры применения типов данных, связанных с датой и временем

Поле применения календарных типов данных слишком велико, чтобы мы смогли объять его целиком. Однако мы можем рассказать о нескольких наиболее популярных операциях, применяющихся в анализе временных рядов, и посмотреть, как они реализуются в Polars. В показанных ниже примерах мы в основном будем работать с типом данных `Date`, но аналогичные операции применимы и для других календарных типов данных.

## *Загрузка из файла CSV*

Для начала загрузим данные, содержащие календарные значения, из файла CSV. Для этого воспользуемся функцией `pl.read_csv()` и передадим ее аргументу `try_parse_dates` значение `True`, чтобы разбор календарных значений выполнялся автоматически:

```
pl.read_csv("data/all_stocks.csv", try_parse_dates=True)
```

Вывод:

```
shape: (18_476, 8)
```

| symbol | date | open | … | close | adj close | volume |
|---|---|---|---|---|---|---|
| str | date | f64 | | f64 | f64 | i64 |
| ASML | 1999-01-04 | 11.765625 | … | 12.140625 | 7.522523 | 1801867 |
| ASML | 1999-01-05 | 11.859375 | … | 13.96875 | 8.655257 | 8241600 |
| ASML | 1999-01-06 | 14.25 | … | 16.875 | 10.456018 | 16400267 |
| ASML | 1999-01-07 | 14.742188 | … | 16.851563 | 10.441495 | 17722133 |
| ASML | 1999-01-08 | 16.078125 | … | 15.796875 | 9.787995 | 10696000 |
| … | … | … | … | … | … | … |
| TSM | 2023-06-26 | 102.019997 | … | 100.110001 | 99.125954 | 8560000 |
| TSM | 2023-06-27 | 101.150002 | … | 102.080002 | 101.076591 | 9732000 |
| TSM | 2023-06-28 | 100.5 | … | 100.919998 | 99.927986 | 8160900 |
| TSM | 2023-06-29 | 101.339996 | … | 100.639999 | 99.650742 | 7383900 |
| TSM | 2023-06-30 | 101.400002 | … | 100.919998 | 99.927986 | 11701700 |

Как видите, данные в столбце `date` были прочитаны в корректном формате.

## *Преобразование в и из типа String*

Также вы можете получить значения календарного типа из строковых значений следующим образом:

```
dates = pl.DataFrame({"date_str": ["2023-12-31", "2024-02-29"]}).with_columns(
    date=pl.col("date_str").str.to_date("%Y-%m-%d")
)

dates
```

Вывод:

```
shape: (2, 2)
┌────────────┬────────────┐
│ date_str   ┆ date       │
│ ---        ┆ ---        │
│ str        ┆ date       │
╞════════════╪════════════╡
│ 2023-12-31 ┆ 2023-12-31 │
│ 2024-02-29 ┆ 2024-02-29 │
└────────────┴────────────┘
```

Чтобы представить даты в текстовом виде определенного формата, можно воспользоваться обратным методом, показанным ниже:

```
dates.with_columns(formatted_date=pl.col("date").dt.to_string("%d-%m-%Y"))
```

Вывод:

```
shape: (2, 3)
┌────────────┬────────────┬────────────────┐
│ date_str   ┆ date       ┆ formatted_date │
│ ---        ┆ ---        ┆ ---            │
│ str        ┆ date       ┆ str            │
╞════════════╪════════════╪════════════════╡
│ 2023-12-31 ┆ 2023-12-31 ┆ 31-12-2023     │
│ 2024-02-29 ┆ 2024-02-29 ┆ 29-02-2024     │
└────────────┴────────────┴────────────────┘
```

Здесь мы методу `Expr.dt.to_string()` передали строку форматирования вида `%d-%m-%Y`, что означает разделение дня, месяца и года в датах при помощи дефисов. С расширенным списком опций можно ознакомиться на страницах официальной документации по адресу *https://docs.rs/chrono/latest/chrono/format/strftime/index.html*.

## *Генерирование диапазонов дат*

Помимо загрузки календарных данных извне, вы можете сгенерировать их вручную прямо в Polars, как показано ниже:

```
pl.DataFrame(
    {
        "monday": pl.date_range(
            start=pl.date(2024, 10, 28),
            end=pl.date(2024, 12, 1),
            interval="1w", ❶
```

```
            eager=True, ❷
        ),
    }
)
```

❶ С помощью аргумента вы можете передать строковое представление интервала между календарными значениями. Например, значение `"1w"` указывает на интервал в одну неделю, `"1d"` – в один день, `"1h"` – в один час и т. д.

❷ Вы можете установить значение аргумента `eager` в `True`, чтобы результат вернулся в виде объекта Series, а значение `False` соответствует возвращению выражения. Поскольку в этом примере мы вызываем конструктор датафрейма, мы не можем использовать выражение, – в этом случае получим ошибку следующего содержания: `TypeError: passing Expr objects to the DataFrame constructor is not supported`.

Вывод:

```
shape: (5, 1)
┌────────────┐
│ monday     │
│ ---        │
│ date       │
╞════════════╡
│ 2024-10-28 │
│ 2024-11-04 │
│ 2024-11-11 │
│ 2024-11-18 │
│ 2024-11-25 │
└────────────┘
```

## *Часовые пояса*

Одной из самых неблагодарных вещей при работе с календарными данными является обработка часовых поясов. Это может стать настоящей головной болью для разработчика. По этой причине зачастую при анализе временных рядов используется так называемое Всемирное координированное время (Coordinated Universal Time – UTC) в качестве фиксированной временной зоны. Далее вы можете переводить ваши календарные значения в нужные вам часовые пояса.

В следующем примере у нас есть датафрейм со значениями UTC, и нам необходимо преобразовать их в часовой пояс Амстердама, т. е. в центральноевропейское время (Central European Time – CEST):

```
pl.DataFrame( ❶
    {
        "utc_mixed_offset": [
            "2021-03-27T00:00:00+0100",
            "2021-03-28T00:00:00+0100",
            "2021-03-29T00:00:00+0200",
            "2021-03-30T00:00:00+0200",
        ]
    }
```

```
).with_columns(
    parsed=pl.col("utc_mixed_offset").str.to_datetime(
        "%Y-%m-%dT%H:%M:%S%z"
    ) ❷
).with_columns(
    converted=pl.col("parsed").dt.convert_time_zone("Europe/Amsterdam") ❸
)
```

❶ Мы создали датафрейм с датами и временем в формате `String`.
❷ Преобразовали типы `String` в `Datetime` с помощью метода `Expr.str.to_datetime()`. Составляющая `%z` при передаче формата говорит о наличии часового пояса в значениях.
❸ Преобразовали полученные значения типа `Datetime` в часовой пояс Амстердама с помощью метода `Expr.dt.convert_time_zone()`.

Вывод:

```
shape: (4, 3)
┌──────────────────────────┬──────────────────────────┬──────────────────────────┐
│ utc_mixed_offset         ┆ parsed                   ┆ converted                │
│ ---                      ┆ ---                      ┆ ---                      │
│ str                      ┆ datetime[μs, UTC]        ┆ datetime[μs,             │
│                          ┆                          ┆ Europe/Amsterdam]        │
╞══════════════════════════╪══════════════════════════╪══════════════════════════╡
│ 2021-03-27T00:00:00+0100 ┆ 2021-03-26 23:00:00 UTC  ┆ 2021-03-27 00:00:00 CET  │
│ 2021-03-28T00:00:00+0100 ┆ 2021-03-27 23:00:00 UTC  ┆ 2021-03-28 00:00:00 CET  │
│ 2021-03-29T00:00:00+0200 ┆ 2021-03-28 22:00:00 UTC  ┆ 2021-03-29 00:00:00 CEST │
│ 2021-03-30T00:00:00+0200 ┆ 2021-03-29 22:00:00 UTC  ┆ 2021-03-30 00:00:00 CEST │
└──────────────────────────┴──────────────────────────┴──────────────────────────┘
```

Как видите, в результирующем датафрейме исходные значения были успешно приведены к заданному часовому поясу. Сам часовой пояс при отображении значений используется в двух видах: CET и CEST, что соответствует летнему центральноевропейскому времени.

В главе 13 мы узнаем, как можно группировать и агрегировать календарные данные с помощью оконных функций, динамической группировки и т. д.

# Тип данных List

Существует три способа хранения коллекций в столбцах датафрейма: с использованием массивов, списков и структур.

В столбце с типом данных *`List`* могут содержаться списки произвольной длины, вмещающие в себя значения одного и того же типа.

**История о двух списках**

Стоит учитывать, что списки в Polars, которые могут хранить элементы одного и того же типа, отличаются от встроенного типа списков в Python, который допускает соседство элементов различных типов. В Polars такого поведения можно достигнуть, ис-

пользуя тип `Object`, чтобы хранить списки из Python, но делать это не рекомендуется, поскольку содержимое в этом случае будет представлять собой двоичные объекты сериализованных данных из Python. Таким образом, при использовании типа `Object` вы не сможете проводить дополнительные манипуляции со списками, а оптимизатор Polars не сможет выполнить свою работу. В результате под капотом все действия будут выполняться на языке Python, а не Rust, что негативно скажется на эффективности.

Тип данных `List` реализован в памяти в виде списка Arrow переменного размера. Подробнее о хранении данных в памяти в Arrow мы будем говорить в главе 18. Подобно типу данных `String`, здесь мы тоже имеем дело с непрерывным буфером данных и буфером смещений, указывающих на расположение значений в буфере данных.

## Методы типа данных List

В табл. 12.9 перечислены методы, предназначенные для работы с типом данных `List`.

***Таблица 12.9. Методы для работы с типом данных*** *`List`*

| Метод | Описание |
|---|---|
| *`Expr.list.all()`* | Определяет, являются ли все булевы значения в списке истинными |
| *`Expr.list.any()`* | Определяет, является ли хотя бы одно булево значение в списках истинным |
| *`Expr.list.arg_max()`* | Извлекает индекс максимального значения в каждом подсписке |
| *`Expr.list.arg_min()`* | Извлекает индекс минимального значения в каждом подсписке |
| *`Expr.list.concat()`* | Объединяет массивы в объекте Series списочного типа |
| *`Expr.list.contains()`* | Проверяет, содержат ли подсписки переданное значение |
| *`Expr.list.count_matches()`* | Подсчитывает количество вхождений произведенного элементом значения |
| *`Expr.list.diff()`* | Подсчитывает разницу между смещенными элементами в каждом подсписке |
| *`Expr.list.drop_nulls()`* | Удаляет пропущенные значения в списке |
| *`Expr.list.eval()`* | Выполняет любое выражение Polars применительно к элементам типа `List` |
| *`Expr.list.explode()`* | Возвращает столбец с отдельными строками для каждого элемента типа `List` |
| *`Expr.list.first()`* | Извлекает первое значение из подсписков |
| *`Expr.list.gather()`* | Получает подсписки по множеству индексов |
| *`Expr.list.gather_every()`* | Получает каждое n-ное значение из подсписков начиная с заданного смещения |
| *`Expr.list.get()`* | Получает значение из подсписков по индексу |
| *`Expr.list.head()`* | Получает срез из первых n элементов из подсписков |
| *`Expr.list.join()`* | Объединяет все строковые элементы в подсписках с заданным разделителем |

***Таблица 12.9*** *(окончание)*

| Метод | Описание |
|---|---|
| *Expr.list.last()* | Извлекает последнее значение из подсписков |
| *Expr.list.len()* | Возвращает количество элементов в каждом списке |
| *Expr.list.max()* | Вычисляет максимальное значение в списках в виде массива |
| *Expr.list.mean()* | Вычисляет среднее значение в списках в виде массива |
| *Expr.list.median()* | Вычисляет медианное значение в списках в виде массива |
| *Expr.list.min()* | Вычисляет минимальное значение в списках в виде массива |
| *Expr.list.n_unique()* | Вычисляет количество уникальных элементов в каждом подсписке |
| *Expr.list.reverse()* | Обращает расположение элементов в списке |
| *Expr.list.sample()* | Извлекает выборку из списка |
| *Expr.list.set_difference()* | Вычисляет различия между элементами двух списков |
| *Expr.list.set_intersection()* | Вычисляет пересечения между элементами двух списков |
| *Expr.list.set_symmetric_difference()* | Вычисляет симметричные различия между элементами двух списков |
| *Expr.list.set_union()* | Создает объединение элементов двух списков |
| *Expr.list.shift()* | Сдвигает значения в списке на заданное количество элементов |
| *Expr.list.slice()* | Извлекает срез из каждого подсписка |
| *Expr.list.sort()* | Сортирует списки в столбце |
| *Expr.list.std()* | Вычисляет стандартное отклонение в списках в виде массива |
| *Expr.list.sum()* | Вычисляет сумму в списках в виде массива |
| *Expr.list.tail()* | Получает срез из последних n элементов из подсписков |
| *Expr.list.to_array()* | Преобразовывает столбец типа `List` в столбец типа `Array` с сохранением исходных типов значений |
| *Expr.list.to_struct()* | Преобразовывает столбец типа `List` в столбец типа `Struct` |
| *Expr.list.unique()* | Извлекает уникальные значения из списка |
| *Expr.list.var()* | Вычисляет дисперсию в списках в виде массива |

# Примеры работы с типом List

Давайте рассмотрим несколько примеров применения методов для работы с типом данных `List`.

Вы можете воспользоваться удобными методами `Expr.list.all()` и `Expr.list.any()` для определения того, являются ли все или хотя бы одно булево значение в списке истинными:

```
bools = pl.DataFrame({"values": [[True, True], [False, False, True], [False]]})

bools.with_columns(
    all_true=pl.col("values").list.all(),
    any_true=pl.col("values").list.any(),
)
```

Вывод:

```
shape: (3, 3)
┌─────────────────────┬──────────┬──────────┐
│ values              ┆ all_true ┆ any_true │
│ ---                 ┆ ---      ┆ ---      │
│ list[bool]          ┆ bool     ┆ bool     │
╞═════════════════════╪══════════╪══════════╡
│ [true, true]        ┆ true     ┆ true     │
│ [false, false, true] ┆ false    ┆ true     │
│ [false]             ┆ false    ┆ false    │
└─────────────────────┴──────────┴──────────┘
```

Мощным методом, который бывает удобно использовать совместно с рассмотренными методами `Expr.list.any()` и `Expr.list.all()`, является `Expr.list.eval()`. Этот метод позволяет выполнить любое выражение Polars применительно к элементам списка. В следующем примере мы воспользуемся методом `Expr.list.eval()` для определения того, все ли перечисленные в списках возрасты превышают 40 лет:

```
groups = pl.DataFrame({"ages": [[18, 21], [30, 40, 50], [42, 69]]})

groups.with_columns(
    over_forty=pl.col("ages").list.eval(
        pl.element() > 40, ❶
        parallel=True, ❷
    )
).with_columns( ❸
    all_over_forty=pl.col("over_forty").list.all() ❹
)
```

❶ Функция *`pl.element()`* используется для осуществления доступа к элементам списка.
❷ Передав аргументу `parallel` значение `True`, мы инициируем выполнение метода `eval()` в параллельном режиме. По умолчанию этот режим отключен, но при включении он может существенно повысить быстродействие, если используемый метод способен выполняться параллельно.
❸ Для обеспечения параллелизма все дальнейшие манипуляции с созданными столбцами должны выполняться в отдельных вызовах метода `with_columns()`.
❹ Метод `Expr.list.all()` применяется для определения того, являются ли все булевы значения в списке истинными.

Вывод:

```
shape: (3, 3)
┌─────────────┬─────────────────────┬──────────────┐
│ ages        ┆ over_forty          ┆ all_over_forty │
│ ---         ┆ ---                 ┆ ---          │
│ list[i64]   ┆ list[bool]          ┆ bool         │
╞═════════════╪═════════════════════╪══════════════╡
│ [18, 21]    ┆ [false, false]      ┆ false        │
│ [30, 40, 50] ┆ [false, false, true] ┆ false        │
│ [42, 69]    ┆ [true, true]        ┆ true         │
└─────────────┴─────────────────────┴──────────────┘
```

Отсортировать элементы в списках вы можете при помощи метода `Expr.list.sort()`, как показано ниже:

```
groups.with_columns(
    ages_sorted_descending=pl.col("ages").list.sort(descending=True)
)
```

Вывод:

```
shape: (3, 2)
┌──────────────┬──────────────────────┐
│ ages         ┆ ages_sorted_descending │
│ ---          ┆ ---                  │
│ list[i64]    ┆ list[i64]            │
╞══════════════╪══════════════════════╡
│ [18, 21]     ┆ [21, 18]             │
│ [30, 40, 50] ┆ [50, 40, 30]         │
│ [42, 69]     ┆ [69, 42]             │
└──────────────┴──────────────────────┘
```

Распаковать списки и разложить их значения по отдельным строкам можно с помощью метода `explode()` следующим образом:

```
groups.explode("ages")
```

Вывод:

```
shape: (7, 1)
┌──────┐
│ ages │
│ ---  │
│ i64  │
╞══════╡
│ 18   │
│ 21   │
│ 30   │
│ 40   │
│ 50   │
│ 42   │
│ 69   │
└──────┘
```

В качестве альтернативы вы можете воспользоваться методом `Expr.list.explode()` или его алиасом `Expr.flatten()`, которые могут быть применены к выражению, как показано ниже:

```
groups.select(ages=pl.col("ages").list.explode())
```

Вывод:

```
shape: (7, 1)
┌──────┐
│ ages │
│ ---  │
```

```
┌─────┐
│ i64 │
╞═════╡
│ 18  │
│ 21  │
│ 30  │
│ 40  │
│ 50  │
│ 42  │
│ 69  │
└─────┘
```

В главе 15 мы продолжим разговор о методе `explode()`.

# Тип данных Array

Тип данных *`Array`* позволяет хранить последовательности фиксированной длины, содержащие значения одного и того же типа. В библиотеке NumPy аналогом этого типа данных является тип `ndarray`.

**Массивы быстрее списков**

Тип данных `Array` внутренне реализован в виде списков Arrow фиксированной длины. Для этого типа так же, как и для типа `List`, используется непрерывный буфер данных, но необходимость в применении буфера смещений отсутствует из-за постоянства длины элементов. Это делает тип данных Array более эффективным с точки зрения работы с памятью, поскольку для загрузки данных требуется меньше операций поиска.

## Методы типа данных Array

В табл. 12.10 перечислены методы, предназначенные для работы с типом данных `Array`.

***Таблица 12.10. Методы для работы с типом данных*** *`Array`*

| Метод | Описание |
|---|---|
| *`Expr.arr.all()`* | Определяет, являются ли все булевы значения в массивах истинными |
| *`Expr.arr.any()`* | Определяет, является ли хотя бы одно булево значение в массивах истинным |
| *`Expr.arr.arg_max()`* | Извлекает индекс максимального значения в каждом подмассиве |
| *`Expr.arr.arg_min()`* | Извлекает индекс минимального значения в каждом подмассиве |
| *`Expr.arr.contains()`* | Проверяет, содержат ли подмассивы переданное значение |
| *`Expr.arr.count_matches()`* | Подсчитывает количество вхождений произведенного элементом значения |

***Таблица 12.10*** *(окончание)*

| Метод | Описание |
|---|---|
| *Expr.arr.explode()* | Возвращает столбец со строками для каждого элемента массива |
| *Expr.arr.first()* | Извлекает первое значение из подмассивов |
| *Expr.arr.get()* | Получает значение из подмассивов по индексу |
| *Expr.arr.join()* | Объединяет все строковые элементы в подмассивах с заданным разделителем |
| *Expr.arr.last()* | Извлекает последнее значение из подмассивов |
| *Expr.arr.max()* | Вычисляет максимальные значения в подмассивах |
| *Expr.arr.median()* | Вычисляет медианные значения в подмассивах |
| *Expr.arr.min()* | Вычисляет минимальные значения в подмассивах |
| *Expr.arr.n_unique()* | Вычисляет количество уникальных элементов в каждом подмассиве |
| *Expr.arr.reverse()* | Обращает расположение элементов в подмассивах |
| *Expr.arr.shift()* | Сдвигает значения в подмассивах на заданное количество элементов |
| *Expr.arr.sort()* | Сортирует массивы в столбце |
| *Expr.arr.std()* | Вычисляет стандартное отклонение в подмассивах |
| *Expr.arr.sum()* | Вычисляет сумму в подмассивах |
| *Expr.arr.to_list()* | Преобразовывает столбец типа `Array` в столбец типа `List` с сохранением исходных типов значений |
| *Expr.arr.to_struct()* | Преобразовывает столбец типа `Array` в столбец типа `Struct` |
| *Expr.arr.unique()* | Извлекает уникальные значения из подмассивов |
| *Expr.arr.var()* | Вычисляет дисперсию в подмассивах |

# Примеры работы с типом Array

Для демонстрации приемов работы с массивами создадим датафрейм со столбцом с типом данных `Array`, в котором будут содержаться значения температуры воздуха в разных городах:

```
events = pl.DataFrame(
    [
        pl.Series(
            "location", ["Paris", "Amsterdam", "Barcelona"], dtype=pl.String
        ),
        pl.Series(
            "temperatures",
            [
                [23, 27, 21, 22, 24, 23, 22],
                [17, 19, 15, 22, 18, 20, 21],
                [30, 32, 28, 29, 34, 33, 31],
            ],
            dtype=pl.Array(pl.Int64, shape=7),
        ),
    ]
)
```

```
events
```

Вывод:

```
shape: (3, 2)
┌──────────┬──────────────┐
│ location ┆ temperatures │
│ ---      ┆ ---          │
│ str      ┆ array[i64, 7] │
╞══════════╪══════════════╡
│ Paris    ┆ [23, 27, … 22] │
│ Amsterdam ┆ [17, 19, … 21] │
│ Barcelona ┆ [30, 32, … 31] │
└──────────┴──────────────┘
```

Посмотрим, как можно применить методы типа данных `Array` `Expr.arr.median()`, `Expr.arr.max()` и `Exp.arr.arg_max()`:

```
events.with_columns(
    median=pl.col("temperatures").arr.median(),
    max=pl.col("temperatures").arr.max(),
    warmest_dow=pl.col("temperatures").arr.arg_max(),
)
```

Вывод:

```
shape: (3, 5)
┌──────────┬──────────────┬────────┬─────┬────────────┐
│ location ┆ temperatures ┆ median ┆ max ┆ warmest_dow │
│ ---      ┆ ---          ┆ ---    ┆ --- ┆ ---        │
│ str      ┆ array[i64, 7] ┆ f64    ┆ i64 ┆ u32        │
╞══════════╪══════════════╪════════╪═════╪════════════╡
│ Paris    ┆ [23, 27, … 22] ┆ 23.0   ┆ 27  ┆ 1          │
│ Amsterdam ┆ [17, 19, … 21] ┆ 19.0   ┆ 22  ┆ 3          │
│ Barcelona ┆ [30, 32, … 31] ┆ 31.0   ┆ 34  ┆ 4          │
└──────────┴──────────────┴────────┴─────┴────────────┘
```

В итоге были созданы столбцы `median` с медианными значениями температуры, `max` – с максимальными значениями и `warmest_dow` – с индексами, соответствующими самым теплым дням для каждого города.

# Тип данных Struct

Тип данных *`Struct`* предназначен для хранения объектов Series, вложенных в другие объекты Series. На уровне строк эти значения можно представить как традиционные словари в Python с ключами и значениями. В качестве ключей используются имена объектов Series, которые называются *полями* (field), а в качестве значений – значения поля для конкретной строки. Поле также описывается определенным типом данных, связанным с ним, который

диктует характер хранящихся значений. Тип данных `Struct` используется в Polars для работы со множеством объектов Series. На рис. 12.1 схематически изображено хранение данных с типом `Struct`.

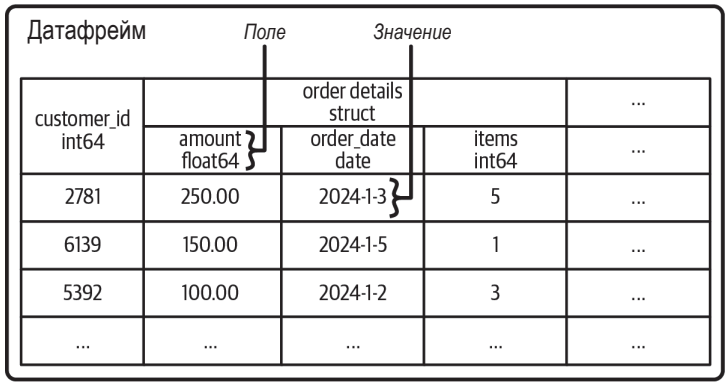



**Рис. 12.1** ❖ Представление типа данных `Struct` в датафрейме

Включая несколько объектов Series в один элемент типа `Struct`, вы не лишаетесь возможности выполнять операции применительно сразу к нескольким столбцам с использованием парадигмы взаимодействия с выражениями. В основном выражения преобразуют объекты Series в новые объекты Series, а тип данных `Struct` позволяет работать сразу с несколькими объектами Series одновременно. Превращение нескольких объектов Series в тип `Struct` не приводит к дублированию данных, а позволяет использовать указатели на существующие буферы данных в памяти, что обеспечивает эффективную работу с подобными структурами.

## Методы типа данных Struct

В табл. 12.11 перечислены методы, предназначенные для работы с типом данных `Struct`.

***Таблица 12.11. Методы для работы с типом данных*** *`Struct`*

| Метод | Описание |
|---|---|
| *`Expr.struct.field()`* | Извлекает поле объекта `Struct` в виде нового объекта Series |
| *`Expr.struct.json_encode()`* | Преобразует объект типа `Struct` в строковый столбец со значениями в формате JSON |
| *`Expr.struct.rename_fields()`* | Переименовывает поля объекта `Struct` |
| *`Expr.struct.unnest()`* | Разворачивает объект типа `Struct` в отдельные поля |
| *`Expr.struct.with_fields()`* | Добавляет или перезаписывает поля объекта `Struct` |

# Примеры работы с типом Struct

Чтобы поработать с типом данных `Struct`, нужно сначала его откуда-то получить. Существует множество методов получения данных в таком формате, но вы можете создать их и вручную с помощью словаря, как показано ниже:

```
from datetime import date

orders = pl.DataFrame(
    {
        "customer_id": [2781, 6139, 5392],
        "order_details": [
            {"amount": 250.00, "date": date(2024, 1, 3), "items": 5},
            {"amount": 150.00, "date": date(2024, 1, 5), "items": 1},
            {"amount": 100.00, "date": date(2024, 1, 2), "items": 3},
        ],
    },
)

orders
```

Вывод:

```
shape: (3, 2)
┌─────────────┬─────────────────────┐
│ customer_id ┆ order_details       │
│ ---         ┆ ---                 │
│ i64         ┆ struct[3]           │
╞═════════════╪═════════════════════╡
│ 2781        ┆ {250.0,2024-01-03,5} │
│ 6139        ┆ {150.0,2024-01-05,1} │
│ 5392        ┆ {100.0,2024-01-02,3} │
└─────────────┴─────────────────────┘
```

**Структура для хранения структур**

Как видите, столбцы с типом данных `Struct` можно создать при помощи списка словарей. Однако по причине колоночного типа хранения объекты `Struct` в памяти содержатся в виде словаря списков (это также видно на рис. 12.1).

Извлечь значения из столбца с типом `Struct` можно с помощью метода `Expr.struct.field()` следующим образом:

```
orders.select(pl.col("order_details").struct.field("amount"))
```

Вывод:

```
shape: (3, 1)
┌────────┐
│ amount │
│ ---    │
│ f64    │
```

```
└────────┘
┌────────┐
│ 250.0  │
│ 150.0  │
│ 100.0  │
└────────┘
```

Для получения результата в виде отдельных столбцов можно воспользоваться методом датафрейма *unnest()* или методом из пространства имен `struct` с именем `Expr.struct.unnest()`, как показано ниже:

```
order_details_df = orders.unnest("order_details")

order_details_df
```

Вывод:

```
shape: (3, 4)
┌─────────────┬────────┬────────────┬───────┐
│ customer_id │ amount │ date       │ items │
│ ---         │ ---    │ ---        │ ---   │
│ i64         │ f64    │ date       │ i64   │
╞═════════════╪════════╪════════════╪═══════╡
│ 2781        │ 250.0  │ 2024-01-03 │ 5     │
│ 6139        │ 150.0  │ 2024-01-05 │ 1     │
│ 5392        │ 100.0  │ 2024-01-02 │ 3     │
└─────────────┴────────┴────────────┴───────┘
```

**Разворачивание и поля**

Применение метода `Expr.struct.unnest()` синонимично использованию инструкции `Expr.struct.field("*")`.

Если вам нужно выполнить обратную операцию и объединить несколько столбцов в один с типом данных `Struct`, вы можете сделать это так:

```
order_details_df.select(
    "amount",
    "date",
    "items",
    order_details=pl.struct(pl.col("amount"), pl.col("date"), pl.col("items")),
)
```

Вывод:

```
shape: (3, 4)
┌────────┬────────────┬───────┬─────────────────────┐
│ amount │ date       │ items │ order_details       │
│ ---    │ ---        │ ---   │ ---                 │
│ f64    │ date       │ i64   │ struct[3]           │
╞════════╪════════════╪═══════╪═════════════════════╡
│ 250.0  │ 2024-01-03 │ 5     │ {250.0,2024-01-03,5} │
│ 150.0  │ 2024-01-05 │ 1     │ {150.0,2024-01-05,1} │
│ 100.0  │ 2024-01-02 │ 3     │ {100.0,2024-01-02,3} │
└────────┴────────────┴───────┴─────────────────────┘
```

Распространенным способом получения объекта типа `Struct` является использование метода `Expr.value_counts()`. Этот метод позволяет подсчитать количество уникальных значений в объекте Series. Метод `Expr.value_counts()` возвращает столбец с типом `Struct` с двумя полями: исходными значениями и количеством.

Сначала создадим датафрейм со столбцом типа `Struct`:

```
basket = pl.DataFrame(
    {
        "fruit": ["cherry", "apple", "banana", "banana", "apple", "banana"],
    }
)

basket
```

Вывод:

```
shape: (6, 1)
┌────────┐
│ fruit  │
│ ---    │
│ str    │
╞════════╡
│ cherry │
│ apple  │
│ banana │
│ banana │
│ apple  │
│ banana │
└────────┘
```

Вы можете подсчитать количество уникальных значений в столбце `fruit` с помощью метода `Expr.value_counts()`, как показано ниже:

```
basket.select(pl.col("fruit").value_counts(sort=True))
```

Вывод:

```
shape: (3, 1)
┌──────────────┐
│ fruit        │
│ ---          │
│ struct[2]    │
╞══════════════╡
│ {"banana",3} │
│ {"apple",2}  │
│ {"cherry",1} │
└──────────────┘
```

Здесь мы вызвали метод `Expr.value_counts()` с аргументом `sort`, которому было передано значение `True`, чтобы получить результаты в упорядоченном виде.

После этого можно развернуть полученный столбец с типом `Struct`, применив к нему метод `Expr.struct.unnest()`:

```
basket.select(pl.col("fruit").value_counts(sort=True).struct.unnest())
```

Вывод:

```
shape: (3, 2)
┌────────┬───────┐
│ fruit  ┆ count │
│ ---    ┆ ---   │
│ str    ┆ u32   │
╞════════╪═══════╡
│ banana ┆ 3     │
│ apple  ┆ 2     │
│ cherry ┆ 1     │
└────────┴───────┘
```

В результате мы получили два отдельных столбца со значениями, избавившись от типа данных `Struct`.

# Заключение

В этой главе мы познакомились с типами данных в Polars, обладающими собственными пространствами имен, и научились с ними работать.

Вы узнали, что:

- текстовые данные удобно хранить при помощи типа `String`, а работать с ними можно посредством богатой палитры методов, представленных в пространстве имен `str`;
- для эффективного хранения текстовых данных с повторяющимися значениями можно воспользоваться специальными типами данных `Categorical` и `Enum`;
- для работы с календарными значениями в Polars применяются типы данных `Date`, `Datetime`, `Time` и `Duration` со своими наборами методов;
- для хранения сложных вложенных данных в Polars можно воспользоваться типами данных `List`, `Array` и `Struct`, каждый из которых обладает собственным пространством имен.

Все эти типы данных предназначены для хранения и работы со значениями, обладающими различной природой и характеристиками.

В следующей главе мы поговорим о группировке и агрегации данных.

# Глава 13

# Группировка и агрегация данных

Группировка и агрегация являются неотъемлемой частью практически любого процесса анализа данных, позволяющего преобразовать сырые необработанные значения в статистически значимые выводы. Работаете ли вы с данными о продажах, информацией о клиентах или показателями датчиков, при помощи агрегации вы сможете ответить на любые вопросы и выявить любые тенденции, которые могут быть скрыты.

Перед аналитиками часто могут стоять следующие вопросы:

- какова чистая выручка в разрезе магазинов?
- сколько отдельных товаров приобрел каждый покупатель?
- какие расходы несет компания в расчете на каждую категорию товаров ежемесячно?

Это именно те вопросы, на которые вы сможете ответить при помощи операций группировки и агрегации данных. Объединяя данные на основе одного или нескольких столбцов и производя вычисления с образовавшимися группами, такие как расчет суммы, среднего, количества и т. д., вы сможете легко обнаружить шаблоны в данных, которые без группировки и агрегации заметить просто невозможно.

В библиотеке Polars операция группировки выполняется посредством метода датафрейма *`group_by()`*, который позволяет объединить данные по одному или нескольким столбцам и измерениям. После выполнения группировки вы можете применить любую агрегирующую функцию для анализа результатов. К примеру, вы можете рассчитать сумму, среднее или медиану для каждой образованной группы или количество строк, попавших в каждую категорию.

В этой главе вы узнаете:

- о контексте GroupBy и доступных для него методах и способах анализа данных;
- о группировке данных на основе календарных значений с использованием специальных методов `group_by_dynamic()`, `rolling()` и `Expr.over()`;

- об оптимизациях, которые доступны разработчикам для повышения эффективности выполнения операций группировки и агрегации данных.

Все инструкции по загрузке нужных файлов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

# Разделяем, применяем и объединяем

В основе группировки и агрегации данных лежит концепция, называемая *Разделяем, применяем и объединяем* (split, apply, combine). Она подразумевает выполнение операций применительно к группам данных. Вот как она работает:

- *разделяем*: сначала мы разбиваем наши данные на группы на основе одного или нескольких столбцов, по которым хотим выделить категории;
- *применяем*: после этого применяем агрегирующую функцию, такую как сумму, среднее или подсчет количества строк, к каждой группе;
- *объединяем*: в заключение собираем результаты агрегации в один датафрейм.

Эта процедура позволяет эффективно выполнять операции над большими наборами данных и получать агрегированные результаты.

В следующем разделе мы посмотрим, как работает метод датафрейма `group_by()`, и разберем несколько полезных примеров.

# Контекст GroupBy

С помощью метода датафрейма *`group_by()`* можно определить, как именно он будет разбит на группы. Рассмотрим следующий пример:

```
fruit = pl.read_csv("data/fruit.csv")
fruit_grouped = fruit.group_by("is_round")
fruit_grouped
```

Вывод:

```
<polars.dataframe.group_by.GroupBy at 0x129732240>
```

Как видите, никакого разбитого датафрейма мы пока не получили. Вместо этого объект `fruit_grouped` представляет собой так называемый *контекст GroupBy* (GroupBy context). По сути, это инструкция для разделения датафрейма на группы, применительно к которой могут быть вызваны методы, перечисленные в табл. 13.1. При вызове одного из этих методов вычисление применяется к каждой группе, после чего результаты вычислений объединя-

ются в итоговый датафрейм. К примеру, для подсчета круглых и некруглых фруктов мы можем воспользоваться методом `len()` следующим образом:

```
fruit_grouped.len()
```

Вывод:

```
shape: (2, 2)
┌──────────┬─────┐
│ is_round ┆ len │
│ ---      ┆ --- │
│ bool     ┆ u32 │
╞══════════╪═════╡
│ false    ┆ 6   │
│ true     ┆ 4   │
└──────────┴─────┘
```

В результате мы получили датафрейм с двумя строками – по числу уникальных значений в столбце `is_round`. В табл. 13.1 приведены все методы, доступные в контексте GroupBy.

***Таблица 13.1. Методы контекста GroupBy***

| Метод | Описание |
|---|---|
| *`GroupBy.__iter__()`* | Позволяет пройти итерациями по группам (вложенным датафреймам), полученным в результате выполнения операции группировки |
| *`GroupBy.agg()`* | Рассчитывает агрегации для каждой группы |
| *`GroupBy.all()`* | Агрегирует группы в объекты Series |
| *`GroupBy.count()`* | Возвращает количество строк в каждой группе |
| *`GroupBy.first()`* | Агрегирует первые значения в группах |
| *`GroupBy.head()`* | Возвращает первые n строк в каждой группе |
| *`GroupBy.last()`* | Агрегирует последние значения в группах |
| *`GroupBy.len()`* | Возвращает количество строк в каждой группе |
| *`GroupBy.map_groups()`* | Применяет пользовательские функции к группам |
| *`GroupBy.max()`* | Сводит группы к подсчету максимального значения |
| *`GroupBy.mean()`* | Сводит группы к подсчету среднего значения |
| *`GroupBy.median()`* | Возвращает медианное значение для каждой группы |
| *`GroupBy.min()`* | Сводит группы к подсчету минимального значения |
| *`GroupBy.n_unique()`* | Возвращает количество уникальных значений в каждой группе |
| *`GroupBy.quantile()`* | Вычисляет заданный квантиль в каждой группе |
| *`GroupBy.sum()`* | Сводит группы к подсчету суммарного значения |
| *`GroupBy.tail()`* | Возвращает последние n строк в каждой группе |

Для дальнейших примеров работы метода `group_by()` давайте загрузим датафрейм из файла *data/top-2000-2023.xlsx*, в котором хранится плейлист ежегодной нидерландской радиопрограммы Top2000. Мы с ним уже работали в главе 6. Напомним, что в этом файле содержится информация о 2000 самых популярных песен по итогам опроса слушателей в 2023 году.

```
top2000 = pl.read_excel(
    "data/top2000-2023.xlsx", read_options={"skip_rows": 1}
).set_sorted("positie")
```

**Не стоит хранить секреты**

Здесь мы воспользовались методом датафрейма *set_sorted()*, чтобы дать Polars понять, что датафрейм уже упорядочен по столбцу `positie`. Нам эта информация просто известна, и мы поделились ей с Polars. Это позволит движку библиотеки применять оптимизации, доступные только для упорядоченных данных. Это один из способов воспользоваться специфическими знаниями о данных, с которыми вы работаете. Но стоит помнить, что если ваши предположения окажутся неверны, Polars может вернуть неправильные результаты.

Посмотреть на значения, к которым будут применяться агрегации, можно, превратив группы в списки при помощи следующего кода:

```
(
    top2000.group_by("jaar")
    .agg( ❶
        songs=pl.concat_str(
            pl.col("artiest"), pl.lit(" - "), pl.col("titel")
        ), ❷
    )
    .sort("jaar", descending=True)
)
```

❶ Метод `GroupBy.agg()` позволяет вам перечислить агрегации, которые вы хотите применить к сгруппированным данным. Мы вернемся к этому методу после обсуждения стандартных агрегаций.

❷ Создаем список композиций путем конкатенации исполнителя и названия песни через дефис.

Вывод:

```
shape: (67, 2)
┌──────┬──────────────────────────────────────────────────┐
│ jaar ┆ songs                                            │
│ ---  ┆ ---                                              │
│ i64  ┆ list[str]                                        │
╞══════╪══════════════════════════════════════════════════╡
│ 2022 ┆ ["Son Mieux - Multicolor", "Bankzitters - Je Blik … │
│ 2021 ┆ ["Goldband - Noodgeval", "Bankzitters - Stapelgek"… │
│ 2020 ┆ ["DI-RECT - Soldier On", "Miss Montreal - Door De … │
│ 2019 ┆ ["Danny Vera - Roller Coaster", "Floor Jansen & He… │
│ 2018 ┆ ["Lady Gaga & Bradley Cooper - Shallow", "White Li… │
│ …    ┆ …                                                │
│ 1960 ┆ ["Etta James - At Last", "Shadows - Apache"]     │
│ 1959 ┆ ["Jacques Brel - Ne Me Quitte Pas", "Elvis Presley… │
│ 1958 ┆ ["Chuck Berry - Johnny B. Goode", "Ella Fitzgerald… │
│ 1957 ┆ ["Johnny Cash - I Walk The Line", "Elvis Presley -… │
│ 1956 ┆ ["Elvis Presley - Love Me Tender", "Elvis Presley … │
└──────┴──────────────────────────────────────────────────┘
```

Воспользовавшись методом `group_by()`, мы смогли легко получить списки песен по годам.

## Примеры применения агрегаций

В следующем фрагменте кода мы получим три первые (наиболее популярные) песни для трех последних лет при помощи метода `GroupBy.head()`:

```
(
    top2000.group_by("jaar", maintain_order=True) ❶
    .head(3) ❷
    .sort("jaar", descending=True)
    .head(9) ❸
)
```

❶ Мы знаем, что наш датафрейм упорядочен по позициям, а значит, можем сохранить сортировку. Передав аргументу `maintain_order` значение `True`, мы изъявляем намерение оставить порядок следования элементов нетронутым. Если оставить значение `False` (используется по умолчанию), порядок следования элементов может быть утрачен по причине параллельной обработки групп.

❷ Нам нужно получить три первые песни в разрезе лет, так что мы можем воспользоваться методом `GroupBy.head()`. Этот метод применяется к контексту GroupBy и как раз приводит к выделению первых трех элементов для каждой группы.

❸ Следующий вызов метода `head(9)` относится уже к датафрейму, а не к контексту GroupBy, и он возвращает первые девять строк итогового датафрейма, что соответствует получению первых трех песен за три года.

Вывод:

```
shape: (9, 4)
┌──────┬─────────┬──────────────────────┬──────────────────┐
│ jaar ┆ positie ┆ titel                ┆ artiest          │
│ ---  ┆ ---     ┆ ---                  ┆ ---              │
│ i64  ┆ i64     ┆ str                  ┆ str              │
╞══════╪═════════╪══════════════════════╪══════════════════╡
│ 2022 ┆ 179     ┆ Multicolor           ┆ Son Mieux        │
│ 2022 ┆ 370     ┆ Je Blik Richting Mij ┆ Bankzitters      │
│ 2022 ┆ 395     ┆ L'enfer              ┆ Stromae          │
│ 2021 ┆ 55      ┆ Noodgeval            ┆ Goldband         │
│ 2021 ┆ 149     ┆ Stapelgek            ┆ Bankzitters      │
│ 2021 ┆ 210     ┆ Dat Heb Jij Gedaan   ┆ Meau             │
│ 2020 ┆ 19      ┆ Soldier On           ┆ DI-RECT          │
│ 2020 ┆ 38      ┆ Door De Wind         ┆ Miss Montreal    │
│ 2020 ┆ 77      ┆ Impossible (Orchestr… ┆ Nothing But Thieves │
└──────┴─────────┴──────────────────────┴──────────────────┘
```

Подобным же образом мы можем получить и три последние (наименее популярные) песни для каждого года, воспользовавшись методом `GroupBy.tail()`, как показано ниже:

```
(
    top2000.group_by("jaar", maintain_order=True)
    .tail(3)
    .sort("jaar", descending=True)
    .head(9)
)
```

Вывод:

```
shape: (9, 4)
┌──────┬─────────┬──────────────────┬──────────────────────┐
│ jaar ┆ positie ┆ titel            ┆ artiest              │
│ ---  ┆ ---     ┆ ---              ┆ ---                  │
│ i64  ┆ i64     ┆ str              ┆ str                  │
╞══════╪═════════╪══════════════════╪══════════════════════╡
│ 2022 ┆ 1391    ┆ De Diepte        ┆ S10                  │
│ 2022 ┆ 1688    ┆ Zeit             ┆ Rammstein            │
│ 2022 ┆ 1716    ┆ THE LONELIEST    ┆ Måneskin             │
│ 2021 ┆ 1865    ┆ Bon Gepakt       ┆ Donnie & Rene Froger │
│ 2021 ┆ 1978    ┆ Hold On          ┆ Armin van Buuren ft…. │
│ 2021 ┆ 2000    ┆ Drivers License  ┆ Olivia Rodrigo       │
│ 2020 ┆ 1824    ┆ Smoorverliefd    ┆ Snelle               │
│ 2020 ┆ 1879    ┆ The Business     ┆ Tiësto               │
│ 2020 ┆ 1902    ┆ Levitating       ┆ Dua Lipa ft. DaBaby  │
└──────┴─────────┴──────────────────┴──────────────────────┘
```

**Первые и последние**

Применение метода `GroupBy.first()` аналогично вызову метода `GroupBy.head(1)`, но для чтения и понимания он проще. Такая же ситуация и с синонимичными вызовами методов `GroupBy.last()` и `GroupBy.tail(1)`.

Теперь выберем десять первых исполнителей по количеству песен в нашем чарте. Это можно сделать, сгруппировав исходные данные по исполнителю и применив агрегацию `GroupBy.len()`, как показано ниже:

```
(top2000.group_by("artiest").len().sort("len", descending=True).head(10))
```

Вывод:

```
shape: (10, 2)
┌────────────────────┬─────┐
│ artiest            ┆ len │
│ ---                ┆ --- │
│ str                ┆ u32 │
╞════════════════════╪═════╡
│ Queen              ┆ 34  │
│ The Beatles        ┆ 31  │
│ ABBA               ┆ 25  │
│ Bruce Springsteen  ┆ 22  │
│ The Rolling Stones ┆ 22  │
│ Coldplay           ┆ 20  │
│ Fleetwood Mac      ┆ 20  │
```

```
| Michael Jackson    | 20  |
| David Bowie        | 18  |
| U2                 | 18  |
└────────────────────┴─────┘
```

Похоже, нидерландцы отдают предпочтение группам Queen, The Beatles и ABBA.

Следующие несколько методов мы покажем на примере другого датафрейма. В нем содержатся данные о продажах. Давайте посмотрим на список столбцов:

```
sales = pl.read_csv("data/sales.csv")
sales.columns
```

Вывод:

```
['Date',
'Day',
'Month',
'Year',
'Customer_Age',
'Age_Group',
'Customer_Gender',
'Country',
'State',
'Product_Category',
'Sub_Category',
'Product',
'Order_Quantity',
'Unit_Cost',
'Unit_Price',
'Profit',
'Cost',
'Revenue']
```

Начнем с демонстрации методов `GroupBy.min()` и `GroupBy.max()`. Скажем, нам необходимо получить самые дорогостоящие категории и подкатегории товаров. Этого можно добиться, сгруппировав данные по категории и подкатегории и затем рассчитав максимальную цену с помощью метода `GroupBy.max()`:

```
(
    sales.select("Product_Category", "Sub_Category", "Unit_Price") ❶
    .group_by("Product_Category", "Sub_Category") ❷
    .max()
    .sort("Unit_Price", descending=True) ❸
    .head(10)
)
```

❶ Выбираем только нужные нам столбцы, чтобы не отвлекаться на прочую информацию.

❷ Группируем данные по категории и подкатегории. В отличие от предыдущего примера, здесь мы применяем группировку сразу к двум столбцам. Это означает, что

метод `GroupBy.max()` вернет нам максимальную цену для всех сочетаний категории и подкатегории товара.

❸ Сортируем результаты по цене в порядке убывания, чтобы извлечь десять самых дорогостоящих категорий.

Вывод:

```
shape: (10, 3)
┌──────────────────┬──────────────────┬────────────┐
│ Product_Category ┆ Sub_Category     ┆ Unit_Price │
│ ---              ┆ ---              ┆ ---        │
│ str              ┆ str              ┆ i64        │
╞══════════════════╪══════════════════╪════════════╡
│ Bikes            ┆ Road Bikes       ┆ 3578       │
│ Bikes            ┆ Mountain Bikes   ┆ 3400       │
│ Bikes            ┆ Touring Bikes    ┆ 2384       │
│ Clothing         ┆ Vests            ┆ 2384       │
│ Accessories      ┆ Bike Stands      ┆ 159        │
│ Accessories      ┆ Bike Racks       ┆ 120        │
│ Clothing         ┆ Shorts           ┆ 70         │
│ Clothing         ┆ Socks            ┆ 70         │
│ Accessories      ┆ Hydration Packs  ┆ 55         │
│ Clothing         ┆ Jerseys          ┆ 54         │
└──────────────────┴──────────────────┴────────────┘
```

Теперь обратимся к информации о суммарном доходе по странам. Для этого сначала сгруппируем данные по странам, после чего применим метод `GroupBy.sum()`, как показано ниже:

```
(
    sales.select("Country", "Profit")
    .group_by("Country")
    .sum()
    .sort("Profit", descending=True)
)
```

Вывод:

```
shape: (6, 2)
┌────────────────┬──────────┐
│ Country        ┆ Profit   │
│ ---            ┆ ---      │
│ str            ┆ i64      │
╞════════════════╪══════════╡
│ United States  ┆ 11073644 │
│ Australia      ┆ 6776030  │
│ United Kingdom ┆ 4413853  │
│ Canada         ┆ 3717296  │
│ Germany        ┆ 3359995  │
│ France         ┆ 2880282  │
└────────────────┴──────────┘
```

А как насчет подкатегорий с наибольшим количеством уникальных товаров? Как вы уже догадались, для получения этой информации нужно сначала сгруппировать данные по подкатегориям, а затем воспользоваться методом `GroupBy.n_unique()`:

```
(
    sales.select("Sub_Category", "Product")
    .group_by("Sub_Category")
    .n_unique()
    .sort("Product", descending=True)
    .head(10)
)
```

Вывод:

```
shape: (10, 2)
┌────────────────────┬─────────┐
│ Sub_Category       │ Product │
│ ---                │ ---     │
│ str                │ u32     │
╞════════════════════╪═════════╡
│ Road Bikes         │ 38      │
│ Mountain Bikes     │ 28      │
│ Touring Bikes      │ 22      │
│ Tires and Tubes    │ 11      │
│ Jerseys            │ 8       │
│ Gloves             │ 4       │
│ Vests              │ 4       │
│ Bottles and Cages  │ 3       │
│ Helmets            │ 3       │
│ Shorts             │ 3       │
└────────────────────┴─────────┘
```

Теперь обратимся к среднему количеству товаров в заказе для каждой возрастной группы покупателей. Для этого сгруппируем данные по столбцу с возрастными группами, после чего получим среднее количество товаров в заказе с помощью метода `GroupBy.mean()`, как показано ниже:

```
(
    sales.select("Age_Group", "Order_Quantity")
    .group_by("Age_Group")
    .mean()
    .sort("Order_Quantity", descending=True)
)
```

Вывод:

```
shape: (4, 2)
┌─────────────┬────────────────┐
│ Age_Group   │ Order_Quantity │
│ ---         │ ---            │
```

```
| str                  | f64           |
╞══════════════════════╪═══════════════╡
| Seniors (64+)        | 13.530137     |
| Youth (<25)          | 12.124018     |
| Adults (35-64)       | 12.045303     |
| Young Adults (25-34) | 11.560899     |
└──────────────────────┴───────────────┘
```

Также вы можете применить агрегирующий метод `GroupBy.quantile()` для получения указанного процентиля по столбцу с доходами для каждой возрастной группы:

```
(
    sales.select("Age_Group", "Revenue")
    .group_by("Age_Group")
    .quantile(0.9)
    .sort("Revenue", descending=True)
)
```

Вывод:

```
shape: (4, 2)
┌──────────────────────┬─────────┐
| Age_Group            | Revenue |
| ---                  | ---     |
| str                  | f64     |
╞══════════════════════╪═════════╡
| Young Adults (25-34) | 2227.0  |
| Adults (35-64)       | 2217.0  |
| Youth (<25)          | 1997.0  |
| Seniors (64+)        | 943.0   |
└──────────────────────┴─────────┘
```

Похоже, молодые профи снова в деле. В Нидерландах специалисты в возрасте от 25 до 34 лет, которых также называют *яппи* (Yuppies), ассоциируются с высокими доходами и большими затратами. Как видите, лидерство по 90-му процентилю в отношении доходов удерживает именно эта категория граждан.

Метод `GroupBy.median()`, предназначенный для получения медианных значений по группам, аналогичен вызову метода `GroupBy.quantile(0.5)`.

Теперь, когда вы познакомились с базовыми агрегациями, доступными для контекста GroupBy, пришло время узнать о нескольких более сложных операциях.

## Методы повышенной сложности

До этого мы обсуждали довольно простые методы агрегации данных. Но иногда появляется необходимость применить более сложные агрегации, выполнить сразу несколько агрегаций или даже воспользоваться собственными

функциями агрегации. И здесь нам на помощь приходит метод *GroupBy.agg()*, о котором мы уже упоминали ранее. Этот метод позволяет:

- агрегировать элементы столбцов в списки для каждой группы;
- управлять именами результирующих столбцов;
- использовать выражения для применения нескольких методов агрегации одновременно и к нескольким столбцам;
- применять собственные функции агрегации посредством выражений.

Давайте пройдемся по всем этим возможностям.

## Агрегация значений в списки

Как мы уже сказали, метод `GroupBy.agg()` делает возможным агрегирование элементов столбцов в списки для каждой группы. Это достигается путем передачи методу `GroupBy.agg()` селектора столбца. Давайте соберем в списки все значения по прибыли (столбец `Profit`) и доходу (столбец `Revenue`) в разрезе стран:

```
(
    sales.select("Country", "Profit", "Revenue")
    .group_by("Country")
    .agg(
        pl.col("Profit"),
        pl.col("Revenue"),
    )
)
```

Вывод:

```
shape: (6, 3)
┌────────────────┬──────────────────────┬──────────────────────┐
│ Country        ┆ Profit               ┆ Revenue              │
│ ---            ┆ ---                  ┆ ---                  │
│ str            ┆ list[i64]            ┆ list[i64]            │
╞════════════════╪══════════════════════╪══════════════════════╡
│ Canada         ┆ [590, 590, … 630]    ┆ [950, 950, … 1014]   │
│ Australia      ┆ [1366, 1188, … 655]  ┆ [2401, 2088, … 1183] │
│ United States  ┆ [524, 407, … 542]    ┆ [929, 722, … 878]    │
│ Germany        ┆ [160, 53, … 746]     ┆ [295, 98, … 1250]    │
│ France         ┆ [427, 427, … 655]    ┆ [787, 787, … 1207]   │
│ United Kingdom ┆ [1053, 1053, … 112]  ┆ [1728, 1728, … 184]  │
└────────────────┴──────────────────────┴──────────────────────┘
```

Именно с этого шага зачастую начинается процедура анализа значений в группах, после чего выбирается агрегирующая функция для полученных групп.

## Переименование столбцов с агрегациями

Также метод `GroupBy.agg()` позволяет присваивать имена результирующим столбцам. Это можно сделать с помощью уже хорошо знакомого вам мето-

да `Expr.alias()` с ключевыми аргументами или посредством специального пространства имен *name*. В табл. 7.2 мы уже перечисляли методы, доступные в этом пространстве имен.

```
(
    sales.group_by("Country").agg(
        pl.col("Profit").alias("All Profits Per Transactions"),
        pl.col("Revenue").name.prefix("All "),
        Cost=pl.col("Revenue") - pl.col("Profit"),
    )
)
```

Вывод:

```
shape: (6, 4)
```

| Country | All Profits Per Tran… | All Revenue | Cost |
|---|---|---|---|
| str | list[i64] | list[i64] | list[i64] |
| Canada | [590, 590, … 630] | [950, 950, … 1014] | [360, 360, … 384] |
| Australia | [1366, 1188, … 655] | [2401, 2088, … 1183] | [1035, 900, … 528] |
| United States | [524, 407, … 542] | [929, 722, … 878] | [405, 315, … 336] |
| Germany | [160, 53, … 746] | [295, 98, … 1250] | [135, 45, … 504] |
| France | [427, 427, … 655] | [787, 787, … 1207] | [360, 360, … 552] |
| United Kingdom | [1053, 1053, … 112] | [1728, 1728, … 184] | [675, 675, … 72] |

## *Одновременное применение нескольких агрегаций*

Кроме того, метод `GroupBy.agg()` позволяет применять одновременно несколько агрегаций и даже не к одному столбцу. Этим можно воспользоваться, передав методу список выражений, как показано ниже:

```
(
    sales.select("Country", "Profit", "Revenue")
    .group_by("Country")
    .agg(
        pl.col("Profit").sum().name.prefix("Total "),
        pl.col("Profit").mean().alias("Average Profit per Transaction"),
        pl.col("Revenue").sum().name.prefix("Total "),
        pl.col("Revenue").mean().alias("Average Revenue per Transaction"),
    )
)
```

Вывод:

```
shape: (6, 5)
```

| Country | Total Profit | Average | Total Revenue | Average |
|---|---|---|---|---|

```
| ---            | ---      | Profit per T… | ---      | Revenue per … |
| str            | i64      | ---           | i64      | ---           |
|                |          | f64           |          | f64           |
╞════════════════╪══════════╪═══════════════╪══════════╪═══════════════╡
| Canada         | 3717296  | 262.187615    | 7935738  | 559.721964    |
| Australia      | 6776030  | 283.089489    | 21302059 | 889.959016    |
| United States  | 11073644 | 282.447687    | 27975547 | 713.552696    |
| Germany        | 3359995  | 302.756803    | 8978596  | 809.028293    |
| France         | 2880282  | 261.891435    | 8432872  | 766.764139    |
| United         | 4413853  | 324.071439    | 10646196 | 781.659031    |
| Kingdom        |          |               |          |               |
└────────────────┴──────────┴───────────────┴──────────┴───────────────┘
```

В качестве альтернативы вы можете использовать селекторы столбцов в комбинации с этими выражениями для применения агрегирующей функции к нескольким столбцам одновременно в одном выражении:

```
(
    sales.select("Country", "Profit", "Revenue")
    .group_by("Country")
    .agg(
        pl.all().sum().name.prefix("Total "),
        pl.all().mean().name.prefix("Average "),
    )
)
```

Вывод:

```
shape: (6, 5)
┌────────────────┬──────────────┬───────────────┬──────────────┬──────────────┐
| Country        | Total Profit | Total Revenue | Average      | Average      |
| ---            | ---          | ---           | Profit       | Revenue      |
| str            | i64          | i64           | ---          | ---          |
|                |              |               | f64          | f64          |
╞════════════════╪══════════════╪═══════════════╪══════════════╪══════════════╡
| Canada         | 3717296      | 7935738       | 262.187615   | 559.721964   |
| Australia      | 6776030      | 21302059      | 283.089489   | 889.959016   |
| United States  | 11073644     | 27975547      | 282.447687   | 713.552696   |
| Germany        | 3359995      | 8978596       | 302.756803   | 809.028293   |
| France         | 2880282      | 8432872       | 261.891435   | 766.764139   |
| United Kingdom | 4413853      | 10646196      | 324.071439   | 781.659031   |
└────────────────┴──────────────┴───────────────┴──────────────┴──────────────┘
```

При использовании выражений вы даже можете применять операторы сравнения, что бывает очень удобно. Оператор сравнения возвращает булев объект Series, в котором истинным результатам сравнения соответствуют значения `True`, а ложным – `False`. Распространенный прием состоит в использовании метода `GroupBy.sum()` для суммирования булевых объектов Series. Это возможно благодаря тому, что значения `True` интерпретируются как единицы, а `False` – как нули. По сути, эта операция сводится к подсчету строк, в которых выполняется условие.

К примеру, мы можем подсчитать количество транзакций с большими значениями в столбце `Profit` (больше 1000) в разрезе стран. Одновременно мы выведем списки с элементами булевых объектов Series для наглядности:

```
(
    sales.select("Country", "Profit")
    .group_by("Country")
    .agg(
        (pl.col("Profit") > 1000).alias("Profit > 1000"),
        (pl.col("Profit") > 1000)
        .sum()
        .alias("Transactions with Profit > 1000"),
    )
)
```

Вывод:

```
shape: (6, 3)
┌────────────────┬──────────────────────────┬──────────────────────────────────┐
│ Country        ┆ Profit > 1000            ┆ Transactions with Profit > 1000 │
│ ---            ┆ ---                      ┆ ---                              │
│ str            ┆ list[bool]               ┆ u32                              │
╞════════════════╪══════════════════════════╪══════════════════════════════════╡
│ Canada         ┆ [false, false, … false] ┆ 868                              │
│ Australia      ┆ [true, true, … false]   ┆ 1233                             │
│ United States  ┆ [false, false, … false] ┆ 2623                             │
│ Germany        ┆ [false, false, … false] ┆ 659                              │
│ France         ┆ [false, false, … false] ┆ 482                              │
│ United Kingdom ┆ [true, true, … false]   ┆ 788                              │
└────────────────┴──────────────────────────┴──────────────────────────────────┘
```

Поскольку вы можете применять в методе `GroupBy.agg()` выражения, ничто не мешает вам воспользоваться функциями Python, возвращающими выражения. Это одно из исключений из правил не использовать движки Polars и Python одновременно. Дело в том, что в этом случае функции Python будут выполняться до Polars, и результатом их работы будут выражения Polars, которые затем будут вычисляться уже в Rust.

В примере, показанном ниже, мы определили функцию Python, принимающую на вход выражение (исходный столбец) и пороговое значение и вычисляющую итоговые выражения на основании выполнения условия превышения значениями в заданном столбце установленного порога:

```
def sum_transactions_above_threshold(
    col: pl.Expr, threshold: float
) -> tuple[pl.Expr, pl.Expr]:
    """Подсчитывает количество транзакций со значениями в столбце, превышающими порог"""
    original_column_name = col.meta.root_names()[0] ❶
    condition_column = (col > threshold).alias(
        f"{original_column_name} > {threshold}"
    )
    new_column = (
```

```
        (col > threshold)
        .sum()
        .alias(f"Transactions with {original_column_name} > {threshold}")
    )
    return condition_column, new_column

sales.select("Country", "Profit").group_by("Country").agg(
    sum_transactions_above_threshold(pl.col("Profit"), 999)
)
```

❶ Метод *meta.root_names()* возвращает имена столбцов, лежащих в основе выражения.

Вывод:

```
shape: (6, 3)
┌────────────────┬──────────────────────┬──────────────────────────────┐
│ Country        ┆ Profit > 999         ┆ Transactions with Profit > 999 │
│ ---            ┆ ---                  ┆ ---                          │
│ str            ┆ list[bool]           ┆ u32                          │
╞════════════════╪══════════════════════╪══════════════════════════════╡
│ Canada         ┆ [false, false, … false] ┆ 868                       │
│ Australia      ┆ [true, true, … false]   ┆ 1233                      │
│ United States  ┆ [false, false, … false] ┆ 2623                      │
│ Germany        ┆ [false, false, … false] ┆ 659                       │
│ France         ┆ [false, false, … false] ┆ 482                       │
│ United Kingdom ┆ [true, true, … false]   ┆ 788                       │
└────────────────┴──────────────────────┴──────────────────────────────┘
```

Возможность использовать пользовательские выражения, как в примере выше, демонстрирует гибкость и универсальность метода `GroupBy.agg()`, и это, в частности, отличает библиотеку Polars от конкурентов.

Существуют и другие способы применять функции, написанные на Python, к вашим данным. Но они относятся к более сложным сценариям, о которых мы поговорим в главе 17.

# Построчные агрегации

Библиотека Polars предлагает богатый набор *горизонтальных агрегаций* (horizontal aggregation), также называемых *построчными*, прямо из коробки. Все они были упомянуты в табл. 9.5 в главе 9. Двумя мощными функциями, позволяющими создавать сложные горизонтальные агрегации, являются *pl.reduce()* и *pl.fold()*. Эти функции работают с целыми объектами Series, зачастую в векторной манере, что делает их столь эффективными.

Давайте посмотрим на примерах, как работают функции `pl.reduce()` и `pl.fold()`. Сначала они создают новую временную колонку, называемую *накопителем* (accumulator). В этой колонке хранятся исходные значения, к которым применяется агрегация. Другим входным параметром является значение,

полученное из выражения после применения агрегации. Этот накопительный столбец обновляется в соответствии с результатом функции, которая принимает в качестве параметров сам накопитель и это значение.

Функции `pl.reduce()` и `pl.fold()` принимают на вход параметры, показанные в табл. 13.2.

***Таблица 13.2. Аргументы функций*** *`pl.reduce()`* ***и*** *`pl.fold()`*

| Аргумент | Описание |
|---|---|
| `Function` | Функция, применяемая к накопителю и обрабатываемому значению |
| `Exprs` | Выражения для применения агрегации |

Также функция `pl.fold()` позволяет указать исходное значение для накопителя с помощью аргумента `acc`, тогда как функция `pl.reduce()` использует в качестве накопителя первое встреченное значение. Посмотрим на пример использования функции `pl.fold()`:

```
fold_example = pl.DataFrame({"col1": [2], "col2": [3], "col3": [4]})

fold_example.with_columns(
    sum=pl.fold(
        acc=pl.lit(0), ❶
        function=lambda acc, x: acc + x, ❷
        exprs=pl.col("*"), ❸
    )
)
```

❶ Поскольку мы хотим просуммировать значения в строке, в качестве начального значения накопителя будем использовать ноль. Функция `pl.reduce()` инициализировала бы накопитель самостоятельно первым значением в столбце.

❷ Здесь мы передали анонимную лямбда-функцию, суммирующую свои аргументы. В ней к накопителю, содержащему результат предыдущей операции, добавляется новое значение из следующего столбца.

❸ Поскольку выражение `pl.col("*")` не накладывает никаких дополнительных фильтров при отборе столбцов, в операции сложения будут участвовать все столбцы без исключения. Если бы мы передали аргументу `exprs` значение `pl.col("col2", "col3")`, в результате получили бы 7, что является итогом сложения значений в указанных столбцах.

Вывод:

```
shape: (1, 4)
┌──────┬──────┬──────┬─────┐
│ col1 ┆ col2 ┆ col3 ┆ sum │
│ ---  ┆ ---  ┆ ---  ┆ --- │
│ i64  ┆ i64  ┆ i64  ┆ i64 │
╞══════╪══════╪══════╪═════╡
│ 2    ┆ 3    ┆ 4    ┆ 9   │
└──────┴──────┴──────┴─────┘
```

Схематически выполнение функции `pl.fold()` показано на рис. 13.1.

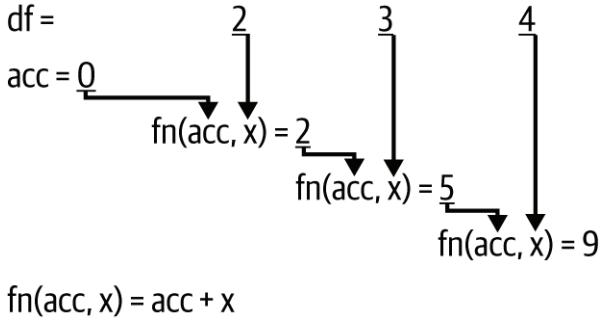



**Рис. 13.1** ❖ Схема работы функции `pl.fold()`

Одним из практических применений функции `pl.fold()` является суммирование значений в столбцах с определенными добавленными весами. К примеру, в нашем датафрейме есть информация о продажах разных товаров, и нам необходимо рассчитать взвешенную сумму продаж. Сделать это можно следующим образом:

```
products = pl.DataFrame(
    {
        "product_A": [10, 20, 30],
        "product_B": [20, 30, 40],
        "product_C": [30, 40, 50],
    }
)

weights = {"product_A": 0.5, "product_B": 1.5, "product_C": 2.0} ❶

weighted_exprs = [ ❷
    (pl.col(product) * weight).alias(product)
        for product, weight in weights.items()
]

products_with_weighted_sum = products.with_columns(
    weighted_sum=pl.fold( ❸
        acc=pl.lit(0), ❹
        function=lambda acc, x: acc + x, ❺
        exprs=weighted_exprs, ❻
    )
)

products_with_weighted_sum
```

❶ Определяем веса для каждого товара.
❷ Создаем выражение в Polars, перемножающее значения столбцов на соответствующие им веса.
❸ Применяем функцию `pl.fold()` для расчета взвешенной суммы.
❹ Инициализируем накопитель нулевым значением.
❺ Передаем функцию суммирования.
❻ Передаем взвешенные выражения в функцию `pl.fold()`.

Вывод:

```
shape: (3, 4)
┌──────────┬──────────┬──────────┬──────────────┐
│ product_A ┆ product_B ┆ product_C ┆ weighted_sum │
│ ---      ┆ ---      ┆ ---      ┆ ---          │
│ i64      ┆ i64      ┆ i64      ┆ f64          │
╞══════════╪══════════╪══════════╪══════════════╡
│ 10       ┆ 20       ┆ 30       ┆ 95.0         │
│ 20       ┆ 30       ┆ 40       ┆ 135.0        │
│ 30       ┆ 40       ┆ 50       ┆ 175.0        │
└──────────┴──────────┴──────────┴──────────────┘
```

# Оконные функции

Иногда вместо агрегирования значений по группам с сокращением количества строк в итоговом датафрейме вам необходимо добавить новую информацию к существующему датафрейму с сохранением количества строк. В таких случаях на помощь приходит метод *`Expr.over()`*. Этот метод позволяет вычислить агрегации применительно к группам в текущем контексте и добавить новую информацию к существующему датафрейму без изменения исходной размерности. Именно этим данная функция отличается от функции группировки. Это бывает полезно, когда вам необходимо сохранить исходные данные в строках и при этом обогатить набор новой информацией, рассчитанной на основе групп. Метод `Expr.over()` обладает аргументами, показанными в табл. 13.3.

***Таблица 13.3. Аргументы метода*** *`Expr.over()`*

| Аргумент | Описание |
|---|---|
| `partition_by` | Столбец или столбцы для группировки. Может принимать результат выражения. Строки воспринимаются как имена столбцов |
| `*more_exprs` | Дополнительные столбцы для группировки, указанные в виде позиционных аргументов |
| `order_by` | Выражение для сортировки результатов применения оконной функции |
| `mapping_strategy` | • `group_to_rows`: если в результате выполнения агрегации получается несколько значений, они присваиваются своим исходным позициям в датафрейме. Это можно сделать только в том случае, если группа до агрегации дает те же элементы, что и после;<br>• `join`: объединяет группы в виде списков и присваивает исходным строкам. Осторожно, эта стратегия может оказаться дорогостоящей с точки зрения расхода памяти;<br>• `explode`: разворачивает сгруппированные данные на новые строки, как при последовательном применении методов `group_by()`, `agg()` и `explode()`. Если сами группы не являются частью операции над окнами, требуется сортировка данных по группам, иначе результат не будет иметь смысла. Эта операция приводит к изменению количества строк |

Давайте вернемся к нашему датафрейму Top2000 из начала главы. Если вы хотите добавить рассчитанные данные к исходному датафрейму, а не агрегировать данные по группам, вы можете воспользоваться методом `Expr.over()`, как показано ниже. Здесь мы вычислим позиции песен внутри каждого года:

```
(
    top2000.select(
        "jaar",
        "artiest",
        "titel",
        "positie",
        year_rank=pl.col("positie").rank().over("jaar"),
    ).sample(10, seed=42)
)
```

Вывод:

```
shape: (10, 5)
```

| jaar | artiest | titel | positie | year_rank |
|---|---|---|---|---|
| i64 | str | str | i64 | f64 |
| 2013 | Stromae | Papaoutai | 318 | 6.0 |
| 1969 | John Denver | Leaving On A Jet Pla… | 607 | 16.0 |
| 1971 | Led Zeppelin | Immigrant Song | 590 | 19.0 |
| 2009 | Anouk | For Bitter Or Worse | 1453 | 23.0 |
| 2015 | Snollebollekes | Links Rechts | 1076 | 14.0 |
| 1984 | Alphaville | Forever Young | 302 | 11.0 |
| 1977 | ABBA | Take A Chance On Me | 636 | 23.0 |
| 1975 | Rod Stewart | Sailing | 918 | 20.0 |
| 1986 | Metallica | Master Of Puppets | 29 | 1.0 |
| 2005 | Alderliefste & Ramse… | Laat Me/Vivre | 463 | 5.0 |

Мы видим, что в 2013 году песня с названием *Papaoutai* исполнителя *Stromae* заняла шестое место, тогда как общая позиция в списке у нее лишь 318-я.

# Динамическая группировка

При работе с календарными данными вам может понадобиться выполнять группировки на основе временных окон. И здесь вам на помощь придет метод датафрейма *`group_by_dynamic()`*. Этот метод вычисляет временные окна с фиксированным шагом и длиной, которым присваивает строки из датафрейма. Это отличается от применения обычного метода `group_by()` тем, что одна и та же строка может оказаться в разных временных окнах в зависимости от их шага и длины. Это бывает полезно при вычислении годовых или

квартальных продаж, когда вам необходимо разделить данные на разные периоды. В табл. 13.4 перечислены аргументы метода `group_by_dynamic()`.

***Таблица 13.4. Аргументы метода*** *`group_by_dynamic()`*

| Аргумент | Описание |
|---|---|
| `Every` | Временной интервал, соответствующий началу каждого окна |
| `Offset` | Используется для указания сдвига начального окна. К примеру, если вы хотите, чтобы ежедневные окна начинались с 9 утра, что соответствует началу рабочего дня, вы можете установить параметры `every=1d` и `offset=9h` |
| `Period` | Длина временного окна. Если параметр не задан, он приравнивается к параметру `every`, что приводит к образованию смежных неперекрывающихся окон. Если же вы хотите создать перекрывающиеся окна, вы можете задать аргументу `period` значение, превышающее значение аргумента `every` |
| `start_by` | Служит для указания стратегии определения начала первого окна. Может соответствовать самому раннему наблюдению, конкретному дню недели или вычисляться на основе самой ранней временной метки со смещением на основе указанного значения аргумента `every` |

Допустимые значения для аргументов `every`, `period` и `offset` перечислены в табл. 13.5.

***Таблица 13.5. Длительности в строковом представлении***

| Длительность | Описание |
|---|---|
| `1ns` | 1 наносекунда |
| `1us` | 1 микросекунда |
| `1ms` | 1 миллисекунда |
| `1s` | 1 секунда |
| `1m` | 1 минута |
| `1h` | 1 час |
| `1d` | 1 календарный день |
| `1w` | 1 календарная неделя |
| `1mo` | 1 календарный месяц |
| `1q` | 1 календарный квартал |
| `1y` | 1 календарный год |
| `1i` | 1 единица индекса |

Перечисленные в табл. 13.5 длительности можно комбинировать. К примеру, значение `"1y6m1w5d"` означает 1 год, 6 месяцев, 1 неделя и 5 дней. Этих параметров достаточно, чтобы создавать разнообразные временные окна и группировать на основе них исходные данные. На рис. 13.2 показаны три разных варианта создания временных окон.

Также при помощи аргумента `closed` вы можете определять, будут ли включаться в группы значения, в точности соответствующие нижней или верхней границе окон. Возможные варианты показаны в табл. 13.6.

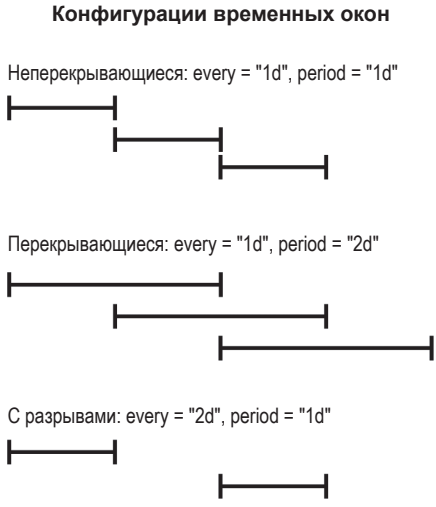



**Рис. 13.2** ❖ Разные варианты создания временных окон

***Таблица 13.6. Допустимые значения аргумента*** *`closed`* ***метода*** *`group_by_dynamic()`*

| Аргумент | Описание | Интервал | Содержит a | Содержит b |
|---|---|---|---|---|
| `Left` | Нижняя граница окна – включительно, верхняя – исключительно | [a, b) | Да | Нет |
| `Right` | Нижняя граница окна – исключительно, верхняя – включительно | (a, b] | Нет | Да |
| `Both` | Обе границы включительно | [a, b] | Да | Да |
| `None` | Обе границы исключительно | (a, b) | Нет | Нет |

# Скользящие агрегации

Если метод `group_by_dynamic()` создает временные окна фиксированного размера, то при помощи метода датафрейма *`rolling()`* можно создавать окна, привязанные к самим значениям в датафрейме. Это бывает полезно при необходимости вычислить скользящие показатели, такие как скользящее среднее или накопительная сумма. Метод `rolling()` принимает аргументы, перечисленные в табл. 13.7.

В датафреймах с временными метками скользящая операция создаст окна для каждой такой метки с расширением назад на указанную в аргументе `period` длительность. При заданном значении аргумента `offset` все окно будет сдвинуто вперед или назад, что позволит сместить фокус аналитики. Это проиллюстрировано на рис. 13.3.

***Таблица 13.7. Аргументы метода*** `rolling()`

| Аргумент | Описание |
|---|---|
| `index_column` | Столбец со значениями, которые будут использоваться как опорные точки при образовании окон |
| `Period` | Размер окна |
| `Offset` | Сдвиг окна вперед или назад |
| `Closed` | Определяет, будут ли включаться в окна значения, в точности соответствующие нижней или верхней границе окон (работает так же точно, как аналогичный аргумент метода `group_by_dynamic()`) |
| `group_by` | Группирует данные по указанным столбцам перед вычислением скользящих агрегаций |

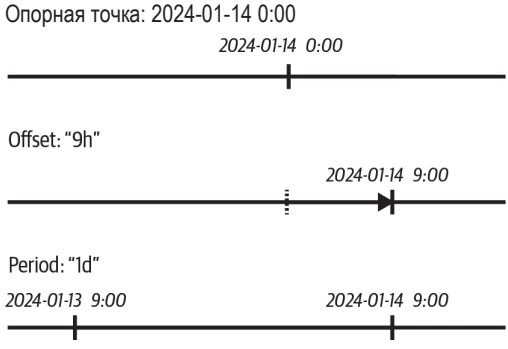



**Рис. 13.3** ❖ Определение окна при использовании метода `rolling()`

Аргумент `group_by` позволяет применить скользящую аналитику в разрезе групп данных.

Представьте, что мы анализируем данные о продажах от сети розничных магазинов и хотим узнать скользящие недельные продажи для каждого магазина в отдельности. Это поможет нам обнаружить тенденции и определить магазины, дающие стабильно лучшие показатели.

Давайте создадим небольшой датафрейм со случайно сгенерированными данными по продажам. Мы разместим в нем данные за несколько недель по двум магазинам, работающим только в будние дни. В дальнейшем мы рассчитаем скользящие недельные суммы продаж по обоим магазинам:

```
dates = pl.date_range( ❶
    start=pl.date(2024, 4, 1),
    end=pl.date(2024, 4, 26),
    interval="2d",
    eager=True, ❷
)
dates = dates.filter(dates.dt.weekday() < 6) ❸
dates_repeated = pl.concat([dates, dates]).sort() ❹

small_sales_df = (
```

```
    pl.DataFrame(
        {
            "date": dates_repeated,
            "store": ["Store A", "Store B"] * dates.len(),
            "sales": [
                200, 150, 220, 160, 250, 180, 270, 190, 280, 210,
                210, 170, 220, 180, 240, 190, 250, 200, 260, 210,
            ],
        }
    )
    .set_sorted("date") ❺
    .set_sorted("store")
)
small_sales_df
```

❶ Создаем диапазон дат с 1 апреля по 25 апреля.
❷ Устанавливаем для аргумента `eager` значение `True`, чтобы датафрейм был создан немедленно.
❸ Исключаем выходные дни.
❹ Повторяем даты для двух магазинов и сортируем их.
❺ Указываем, что датафрейм отсортирован по столбцам `date` и `store`.

Вывод:

```
shape: (20, 3)
┌────────────┬─────────┬───────┐
| date       | store   | sales |
| ---        | ---     | ---   |
| date       | str     | i64   |
╞════════════╪═════════╪═══════╡
| 2024-04-01 | Store A | 200   |
| 2024-04-01 | Store B | 150   |
| 2024-04-03 | Store A | 220   |
| 2024-04-03 | Store B | 160   |
| 2024-04-05 | Store A | 250   |
| …          | …       | …     |
| 2024-04-19 | Store B | 190   |
| 2024-04-23 | Store A | 250   |
| 2024-04-23 | Store B | 200   |
| 2024-04-25 | Store A | 260   |
| 2024-04-25 | Store B | 210   |
└────────────┴─────────┴───────┘
```

Теперь мы можем рассчитать скользящие недельные суммы продаж по обоим магазинам, как показано ниже:

```
result = small_sales_df.rolling( ❶
    index_column="date",
    period="7d",
    group_by="store",
).agg( ❷
```

```
    sum_of_last_7_days_sales=pl.sum("sales")
)

final_df = small_sales_df.join(result, on=["date", "store"]) ❸

final_df
```

❶ Метод `rolling()` создает окна, включающие в себя текущую строку и строки, принадлежащие последним семи дням для каждого из двух магазинов.
❷ Вычисляем сумму для созданных методом `rolling()` скользящих окон.
❸ Присоединяем полученные данные обратно к исходному датафрейму.

Вывод:

```
shape: (20, 4)
┌────────────┬─────────┬───────┬──────────────────────────┐
│ date       ┆ store   ┆ sales ┆ sum_of_last_7_days_sales │
│ ---        ┆ ---     ┆ ---   ┆ ---                      │
│ date       ┆ str     ┆ i64   ┆ i64                      │
╞════════════╪═════════╪═══════╪══════════════════════════╡
│ 2024-04-01 ┆ Store A ┆ 200   ┆ 200                      │
│ 2024-04-03 ┆ Store A ┆ 220   ┆ 420                      │
│ 2024-04-05 ┆ Store A ┆ 250   ┆ 670                      │
│ 2024-04-09 ┆ Store A ┆ 270   ┆ 740                      │
│ 2024-04-11 ┆ Store A ┆ 280   ┆ 800                      │
│ 2024-04-15 ┆ Store A ┆ 210   ┆ 760                      │
│ 2024-04-17 ┆ Store A ┆ 220   ┆ 710                      │
│ 2024-04-19 ┆ Store A ┆ 240   ┆ 670                      │
│ 2024-04-23 ┆ Store A ┆ 250   ┆ 710                      │
│ 2024-04-25 ┆ Store A ┆ 260   ┆ 750                      │
│ 2024-04-01 ┆ Store B ┆ 150   ┆ 150                      │
│ 2024-04-03 ┆ Store B ┆ 160   ┆ 310                      │
│ 2024-04-05 ┆ Store B ┆ 180   ┆ 490                      │
│ 2024-04-09 ┆ Store B ┆ 190   ┆ 530                      │
│ 2024-04-11 ┆ Store B ┆ 210   ┆ 580                      │
│ 2024-04-15 ┆ Store B ┆ 170   ┆ 570                      │
│ 2024-04-17 ┆ Store B ┆ 180   ┆ 560                      │
│ 2024-04-19 ┆ Store B ┆ 190   ┆ 540                      │
│ 2024-04-23 ┆ Store B ┆ 200   ┆ 570                      │
│ 2024-04-25 ┆ Store B ┆ 210   ┆ 600                      │
└────────────┴─────────┴───────┴──────────────────────────┘
```

Как видите, скользящие продажи были вычислены для каждого магазина в отдельности. Первые несколько значений в новом столбце для магазина включают в себя текущую запись и все предыдущие дни, после чего записи, не входящие в текущую неделю, начинают исключаться, чтобы в скользящем показателе учитывались только последние семь календарных дней, как было заявлено при определении окна. Скользящие показатели помогают определить более сглаженные тренды, не подверженные резким перепадам.

# Передискретизация данных

Операция, обратная агрегированию календарных данных, реализуется в Polars при помощи метода датафрейма *upsample()*. Если в вашем временном ряду содержатся разрывы, вы можете воспользоваться этим методом для заполнения пропусков. В табл. 13.8 перечислены аргументы, принимаемые методом upsample().

***Таблица 13.8. Аргументы метода*** *upsample()*

| Аргумент | Описание |
|---|---|
| time_column | Столбец временного ряда. Обратите внимание, что данные должны быть отсортированы по этому столбцу, чтобы вывод имел смысл |
| Every | Дискретность данных |
| group_by | Сначала данные группируются по этому столбцу, после чего для каждой группы производится передискретизация |
| maintain_order | Сохраняет предсказуемый порядок следования строк. Может замедлить работу метода |

Как вы помните, в нашем предыдущем наборе данных присутствовали пропуски дат. Восполнить их можно следующим образом:

```
upsampled_small_sales_df = small_sales_df.upsample(
    time_column="date", every="1d", group_by="store", maintain_order=True
)

upsampled_small_sales_df
```

Вывод:

```
shape: (50, 3)
┌────────────┬─────────┬───────┐
| date       | store   | sales |
| ---        | ---     | ---   |
| date       | str     | i64   |
╞════════════╪═════════╪═══════╡
| 2024-04-01 | Store A | 200   |
| 2024-04-02 | null    | null  |
| 2024-04-03 | Store A | 220   |
| 2024-04-04 | null    | null  |
| 2024-04-05 | Store A | 250   |
| 2024-04-06 | null    | null  |
| 2024-04-07 | null    | null  |
| 2024-04-08 | null    | null  |
| 2024-04-09 | Store A | 270   |
| 2024-04-10 | null    | null  |
| …          | …       | …     |
```

```
| 2024-04-16 | null    | null  |
| 2024-04-17 | Store B | 180   |
| 2024-04-18 | null    | null  |
| 2024-04-19 | Store B | 190   |
| 2024-04-20 | null    | null  |
| 2024-04-21 | null    | null  |
| 2024-04-22 | null    | null  |
| 2024-04-23 | Store B | 200   |
| 2024-04-24 | null    | null  |
| 2024-04-25 | Store B | 210   |
└────────────┴─────────┴───────┘
```

Как видите, мы заполнили пропуски в датах, но напротив них теперь стоят значения `null`, от которых, по-хорошему, нужно избавиться. В главе 8 мы подробно освещали способы для заполнения пропущенных значений. В данном случае мы применим заполнение прямым проходом для магазинов и интерполяцию – для продаж:

```
upsampled_small_sales_df.select(
    "date", pl.col("store").forward_fill(), pl.col("sales").interpolate()
)
```

Вывод:

```
shape: (50, 3)
┌────────────┬─────────┬───────┐
| date       | store   | sales |
| ---        | ---     | ---   |
| date       | str     | f64   |
╞════════════╪═════════╪═══════╡
| 2024-04-01 | Store A | 200.0 |
| 2024-04-02 | Store A | 210.0 |
| 2024-04-03 | Store A | 220.0 |
| 2024-04-04 | Store A | 235.0 |
| 2024-04-05 | Store A | 250.0 |
| 2024-04-06 | Store A | 255.0 |
| 2024-04-07 | Store A | 260.0 |
| 2024-04-08 | Store A | 265.0 |
| 2024-04-09 | Store A | 270.0 |
| 2024-04-10 | Store A | 275.0 |
| …          | …       | …     |
| 2024-04-16 | Store B | 175.0 |
| 2024-04-17 | Store B | 180.0 |
| 2024-04-18 | Store B | 185.0 |
| 2024-04-19 | Store B | 190.0 |
| 2024-04-20 | Store B | 192.5 |
| 2024-04-21 | Store B | 195.0 |
| 2024-04-22 | Store B | 197.5 |
| 2024-04-23 | Store B | 200.0 |
| 2024-04-24 | Store B | 205.0 |
| 2024-04-25 | Store B | 210.0 |
└────────────┴─────────┴───────┘
```

Таким образом вы можете восстанавливать дискретность ваших временных рядов и заполнять пропуски в данных.

# Заключение

В этой главе вы научились группировать и агрегировать данные. Вы узнали, что:

- основные методы агрегации, применимые к контексту `GroupBy`, – это `sum()`, `mean()`, `quantile()` и `median()`;
- способы продвинутой агрегации доступны при использовании метода `GroupBy.agg()`, который позволяет агрегировать элементы столбцов в списки по группам, управлять именами новых столбцов и применять множество функций агрегации одновременно, не ограничиваясь при этом только одним столбцом;
- оконные функции в Polars реализуются посредством метода `Expr.over()`;
- создавать группы на основе временных окон можно при помощи метода датафрейма `group_by_dynamic()`;
- скользящие показатели в датафреймах можно вычислять с помощью метода датафрейма `rolling()`;
- посредством метода датафрейма `set_sorted()` можно дать движку Polars понять, что датафрейм уже отсортирован по одному или нескольким столбцам;
- метод `upsample()` позволит вам передискретизировать временные ряды, заполнив пропуски.

В следующей главе мы будем говорить об объединении датафреймов.

# Глава 14

# Объединение и слияние

Зачастую данные приходят к нам из разных источников, и впоследствии нам приходится их как-то объединять и комбинировать. Существует множество способов сделать это в Polars, и обо всех них мы поговорим в этой главе.

Любопытно то, что именно с объединения данных начиналась библиотека Polars. Столкнувшись с необходимостью выполнить слияние двух файлов CSV в Rust, будущий создатель Polars Ричи Винк задумался о том, чему в итоге посвящена вся эта книга. Это не может не сказываться на настроении, с которым мы пишем эту главу.

В этой главе вы узнаете:

- об использовании метода `join()` для объединения двух датафреймов по заданным полям и с определенной стратегией;
- что можно воспользоваться методом `join_asof()` с целью объединения датафреймов по ближайшим значениям;
- об эффективном применении полезных функций и методов, таких как `pl.concat()`, `vstack()`, `hstack()` и `extend()`;
- как комбинировать объекты Series при помощи метода `append()`;
- о различиях между всеми этими методами и областях их применения.

Все инструкции по загрузке и установке требуемых файлов вы найдете в главе 2. Мы будем исходить из предположения о том, что все рабочие файлы вы поместили в директорию *data*.

## Объединение данных

Для объединения двух датафреймов в библиотеке Polars присутствует метод *`join()`*. Он принимает аргументы, показанные в табл. 14.1.

***Таблица 14.1. Аргументы метода*** *join()*

| Аргумент | Описание |
|---|---|
| Other | Датафрейм, с которым выполняется объединение |
| On | Имена столбцов, по которым будет выполняться объединение, если в обоих датафреймах эти столбцы носят одно имя |
| left_on и right_on | Имена столбцов, по которым будет выполняться объединение, если в разных датафреймах эти столбцы называются по-разному |
| How | Используемая стратегия объединения |
| Suffix | Суффикс, который будет добавляться к именам столбцов, присутствующих в обоих датафреймах |
| Validate | Проверка того, что выполняется объединение определенного типа |
| join_nulls | Объединение значений null. По умолчанию значения null не объединяются |

# Стратегии объединения

Объединение датафреймов может производиться с использованием разных стратегий в зависимости от потребностей. Стратегии, поддерживаемые методом join():

- inner (по умолчанию): сохраняются только строки, присутствующие в обоих датафреймах;
- left: сохраняются все строки из левого датафрейма и только совпадающие – из правого;
- right: сохраняются все строки из правого датафрейма и только совпадающие – из левого;
- full: сохраняются все строки из обоих датафреймов;
- cross: приводит к образованию декартова произведения двух датафреймов. Этот термин из теории множеств означает все возможные комбинации из двух наборов данных. Мы увидим пример далее в этой главе;
- semi: сохраняются все строки из левого датафрейма, имеющие совпадения с правым;
- anti: сохраняются все строки из левого датафрейма, не имеющие совпадений с правым.

В этом разделе мы будем работать с представленными ниже простыми датафреймами:

```
df_left = pl.DataFrame({"key": ["A", "B", "C", "D"], "value": [1, 2, 3, 4]})

df_right = pl.DataFrame({"key": ["B", "C", "D", "E"], "value": [5, 6, 7, 8]})
```

## *Внутреннее объединение*

Стратегией по умолчанию в Polars является *внутреннее объединение* (inner join) датафреймов. При его использовании из обоих датафреймов в итоговый

набор попадают только те строки, которые имеют совпадения по ключевым столбцам. В примере, показанном ниже, видно, что строка из левого датафрейма с ключом `A` не попала в итоговый набор, как и строка из правого датафрейма с ключом `E`. Все остальные строки сохранились, поскольку имеют совпадения по используемому ключу (столбец `key`):

```
df_left.join(df_right, on="key", how="inner")
```

Вывод:

```
shape: (3, 3)
┌─────┬───────┬─────────────┐
│ key ┆ value ┆ value_right │
│ --- ┆ ---   ┆ ---         │
│ str ┆ i64   ┆ i64         │
╞═════╪═══════╪═════════════╡
│ B   ┆ 2     ┆ 5           │
│ C   ┆ 3     ┆ 6           │
│ D   ┆ 4     ┆ 7           │
└─────┴───────┴─────────────┘
```

## *Полное объединение*

При выполнении *полного объединения* (full join) в итоговом наборе данных сохраняются все без исключения строки из обоих датафреймов, а недостающие значения в столбцах заполняются значениями `null`. Вы также можете передать суффикс, который будет добавляться к именам столбцов из правого датафрейма в случае совпадения имен столбцов в обоих датафреймах. В примере ниже мы передали методу соответствующий аргумент со значением `"_other"`:

```
df_left.join(df_right, on="key", how="full", suffix="_other")
```

Вывод:

```
shape: (5, 4)
┌──────┬───────┬───────────┬─────────────┐
│ key  ┆ value ┆ key_other ┆ value_other │
│ ---  ┆ ---   ┆ ---       ┆ ---         │
│ str  ┆ i64   ┆ str       ┆ i64         │
╞══════╪═══════╪═══════════╪═════════════╡
│ B    ┆ 2     ┆ B         ┆ 5           │
│ C    ┆ 3     ┆ C         ┆ 6           │
│ D    ┆ 4     ┆ D         ┆ 7           │
│ null ┆ null  ┆ E         ┆ 8           │
│ A    ┆ 1     ┆ null      ┆ null        │
└──────┴───────┴───────────┴─────────────┘
```

## Левое объединение

*Левое объединение* (left join) подразумевает сохранение в итоговом датафрейме всех строк из левого датафрейма с добавлением значений из строк правого датафрейма, имеющих совпадение по ключевым столбцам. Пустующие значения в этом случае заполняются значениями `null`:

```
df_left.join(df_right, on="key", how="left")
```

Вывод:

```
shape: (4, 3)
┌─────┬───────┬─────────────┐
│ key ┆ value ┆ value_right │
│ --- ┆ ---   ┆ ---         │
│ str ┆ i64   ┆ i64         │
╞═════╪═══════╪═════════════╡
│ A   ┆ 1     ┆ null        │
│ B   ┆ 2     ┆ 5           │
│ C   ┆ 3     ┆ 6           │
│ D   ┆ 4     ┆ 7           │
└─────┴───────┴─────────────┘
```

**Всем оставаться на своих местах!**

При использовании левого объединения порядок строк в итоговом датафрейме будет соответствовать порядку в левом датафрейме.

## Правое объединение

*Правое объединение* (right join) противоположно левому. В результат попадают все строки из правого датафрейма и только совпадающие – из левого:

```
df_left.join(df_right, on="key", how="right")
```

Вывод:

```
shape: (4, 3)
┌───────┬─────┬─────────────┐
│ value ┆ key ┆ value_right │
│ ---   ┆ --- ┆ ---         │
│ i64   ┆ str ┆ i64         │
╞═══════╪═════╪═════════════╡
│ 2     ┆ B   ┆ 5           │
│ 3     ┆ C   ┆ 6           │
│ 4     ┆ D   ┆ 7           │
│ null  ┆ E   ┆ 8           │
└───────┴─────┴─────────────┘
```

## Перекрестное объединение

При выполнении *перекрестного объединения* (cross join) создается так называемое декартово произведение двух датафреймов. Это означает, что количество строк в итоговом датафрейме будет равно произведению размеров двух датафреймов, что может приводить к сильному раздуванию данных. Аргумент `on` в этом случае не нужен, поскольку в любом случае все строки из левого датафрейма будут объединяться со всеми из правого, как показано ниже:

```
df_left.join(df_right, how="cross")
```

Вывод:

```
shape: (16, 4)
┌─────┬───────┬───────────┬─────────────┐
│ key ┆ value ┆ key_right ┆ value_right │
│ --- ┆ ---   ┆ ---       ┆ ---         │
│ str ┆ i64   ┆ str       ┆ i64         │
╞═════╪═══════╪═══════════╪═════════════╡
│ A   ┆ 1     ┆ B         ┆ 5           │
│ A   ┆ 1     ┆ C         ┆ 6           │
│ A   ┆ 1     ┆ D         ┆ 7           │
│ A   ┆ 1     ┆ E         ┆ 8           │
│ B   ┆ 2     ┆ B         ┆ 5           │
│ …   ┆ …     ┆ …         ┆ …           │
│ C   ┆ 3     ┆ E         ┆ 8           │
│ D   ┆ 4     ┆ B         ┆ 5           │
│ D   ┆ 4     ┆ C         ┆ 6           │
│ D   ┆ 4     ┆ D         ┆ 7           │
│ D   ┆ 4     ┆ E         ┆ 8           │
└─────┴───────┴───────────┴─────────────┘
```

## Полуобъединение

*Полуобъединение*, или полусоединение (semi join), представляет собой особый вид связи, при которой значения из правого датафрейма не добавляются к строкам из левого. Вместо этого мы просто сохраняем все строки из левого датафрейма, имеющие совпадения с правым. Таким образом, полусоединение можно рассматривать как дополнительный способ фильтрации левого датафрейма:

```
df_left.join(df_right, on="key", how="semi")
```

Вывод:

```
shape: (3, 2)
┌─────┬───────┐
│ key ┆ value │
│ --- ┆ ---   │
│ str ┆ i64   │
╞═════╪═══════╡
```

```
| B   | 2     |
| C   | 3     |
| D   | 4     |
└─────┴───────┘
```

### *Антиобъединение*

*Антиобъединение*, или антисоединение (anti join), является обратным по отношению к полуобъединению. В результате его применения в итоговом наборе останутся лишь строки из левого датафрейма, не имеющие совпадений с правым, как показано ниже:

```
df_left.join(df_right, on="key", how="anti")
```

Вывод:

```
shape: (1, 2)
┌─────┬───────┐
| key | value |
| --- | ---   |
| str | i64   |
╞═════╪═══════╡
| A   | 1     |
└─────┴───────┘
```

## Объединение по нескольким столбцам

Вы можете объединять датафреймы сразу по нескольким столбцам, для чего необходимо передать аргументу `on` список имен объединяемых столбцов. В результате при поиске соответствий будут учитываться значения во всех перечисленных столбцах.

Для демонстрации этого вида объединения нам понадобятся датафреймы с достаточным количеством столбцов. Мы будем использовать показанные ниже датафреймы, а объединение будем выполнять по столбцам `name` и `city`:

```
residences_left = pl.DataFrame(
    {
        "name": ["Alice", "Bob", "Charlie", "Dave"],
        "city": ["NY", "LA", "NY", "SF"],
        "age": [25, 30, 35, 40],
    }
)

departments_right = pl.DataFrame(
    {
        "name": ["Alice", "Bob", "Charlie", "Dave"],
        "city": ["NY", "LA", "NY", "Chicago"],
        "department": ["Finance", "Marketing", "Engineering", "Operations"],
    }
)
```

```
residences_left.join(departments_right, on=["name", "city"], how="inner")
```

Вывод:

```
shape: (3, 4)
┌─────────┬──────┬─────┬─────────────┐
│ name    ┆ city ┆ age ┆ department  │
│ ---     ┆ ---  ┆ --- ┆ ---         │
│ str     ┆ str  ┆ i64 ┆ str         │
╞═════════╪══════╪═════╪═════════════╡
│ Alice   ┆ NY   ┆ 25  ┆ Finance     │
│ Bob     ┆ LA   ┆ 30  ┆ Marketing   │
│ Charlie ┆ NY   ┆ 35  ┆ Engineering │
└─────────┴──────┴─────┴─────────────┘
```

# Проверка объединения

После объединения данных вы можете проверить *кардинальность* (cardinality) результата. Под кардинальностью имеется в виду количество экземпляров одной сущности, которые могут быть связаны с экземплярами другой сущности. В Polars можно выполнить проверку связи на удовлетворение перечисленным ниже типам кардинальности.

## *Многие ко многим*

Тип связи *многие ко многим* (many-to-many – m:m) подразумевает, что строки в обоих датафреймах могут соответствовать нескольким строкам в другом датафрейме. Примером может быть связь между таблицами с сотрудниками и проектами. Каждый сотрудник может быть вовлечен в работу с несколькими проектами, и в то же время над каждым проектом могут работать несколько сотрудников. В Polars это вид проверки по умолчанию, при котором фактически никакие проверки не выполняются.

## *Один ко многим*

Тип связи *один ко многим* (one-to-many – 1:m) предполагает, что строки в левом датафрейме могут соответствовать нескольким строкам в правом датафрейме. Пример – таблицы отделов и сотрудников. В одном отделе могут работать несколько сотрудников, но каждый сотрудник может работать только в одном отделе. При использовании этого типа проверки движок Polars удостоверяется в том, что в левом датафрейме значения, по которым выполняется объединение, являются уникальными.

## *Многие к одному*

Тип связи *многие к одному* (many-to-one – m:1) предполагает, что строки в правом датафрейме могут соответствовать нескольким строкам в левом

датафрейме. Пример – таблицы сотрудников и городов, в которых они проживают. В одном городе могут проживать несколько сотрудников, но каждый сотрудник может проживать только в одном городе. В этом случае Polars проверяет, что в правом датафрейме значения, по которым выполняется объединение, являются уникальными.

### *Один к одному*

Тип связи *один к одному* (one-to-one – 1:1) означает, что каждая строка в одном датафрейме может соответствовать только одной строке в другом датафрейме. Пример – таблицы сотрудников и их идентификаторов. При использовании этого типа проверки движок Polars удостоверяется в том, что в обоих датафреймах значения, по которым выполняется объединение, являются уникальными.

Для выполнения одного из перечисленных выше типов проверки кардинальности необходимо передать соответствующее значение параметру `validate` метода `join()`. Этот параметр может принимать одно из следующих значений: `"m:m"`, `"m:1"`, `"1:m"`, `"1:1"`.

В показанном ниже примере мы объединим датафрейм с сотрудниками с датафреймом с отделами с кардинальностью многие к одному:

```
employees = pl.DataFrame(
    {
        "employee_id": [1, 2, 3, 4],
        "name": ["Alice", "Bob", "Charlie", "Dave"],
        "department_id": [10, 10, 30, 10],
    }
)

departments = pl.DataFrame(
    {
        "department_id": [10, 20, 30],
        "department_name": [
            "Information Technology",
            "Finance",
            "Human Resources",
        ],
    }
)

employees.join(departments, on="department_id", how="left", validate="m:1")
```

Вывод:

```
shape: (4, 4)
┌─────────────┬─────────┬───────────────┬──────────────────────┐
│ employee_id ┆ name    ┆ department_id ┆ department_name      │
│ ---         ┆ ---     ┆ ---           ┆ ---                  │
│ i64         ┆ str     ┆ i64           ┆ str                  │
```

```
┌─────────────┬─────────┬─────────────┬──────────────────────┐
│ 1           │ Alice   │ 10          │ Information Technolo… │
│ 2           │ Bob     │ 10          │ Information Technolo… │
│ 3           │ Charlie │ 30          │ Human Resources       │
│ 4           │ Dave    │ 10          │ Information Technolo… │
└─────────────┴─────────┴─────────────┴──────────────────────┘
```

Если окажется так, что у разных отделов будут одинаковые идентификаторы, проверка завершится неудачей, и вы тут же об этом узнаете. Это показано в примере ниже:

```
departments = pl.DataFrame(
    {
        "department_id": [10, 20, 10],
        "department_name": [
            "Information Technology",
            "Finance",
            "Human Resources",
        ],
    }
)

employees.join(departments, on="department_id", how="left", validate="m:1")
```

Вывод:

```
ComputeError: join keys did not fulfill m:1 validation
```

# Неточное объединение

При объединении датафреймов вам может понадобиться сопоставить значения в ключевых полях на основе неточного соответствия. К примеру, вы можете столкнуться с необходимостью объединения двух таблиц с продажами из разных источников, в одном из которых время фиксируется в момент записи продажи в базу данных, а в другом – в момент произведения оплаты. В результате в таблицах могут появиться некритичные расхождения, которые можно устранить при помощи объединения с неточным соответствием. В Polars это реализуется при помощи метода датафрейма *`join_asof()`*. В табл. 14.2 перечислены допустимые для этого метода аргументы.

***Таблица 14.2. Аргументы метода*** *`join_asof()`*

| Аргумент | Описание |
|---|---|
| `Other` | Датафрейм, с которым выполняется объединение |
| `On` | Имена столбцов, по которым будет выполняться объединение, если в обоих датафреймах эти столбцы носят одно имя |
| `left_on` и `right_on` | Имена столбцов, по которым будет выполняться объединение, если в разных датафреймах эти столбцы называются по-разному |

***Таблица 14.2*** *(окончание)*

| Аргумент | Описание |
|---|---|
| `By` | Столбцы, по которым нужно выполнить объединение перед выполнением метода `join_asof()` |
| `by_left` и `by_right` | Столбцы, по которым нужно выполнить объединение перед выполнением метода `join_asof()`, если в разных датафреймах эти столбцы называются по-разному |
| `Strategy` | Стратегия объединения |
| `Suffix` | Суффикс, который будет добавляться к именам столбцов, присутствующих в обоих датафреймах |
| `Tolerance` | Максимальное расхождение между значениями для признания их совпадения |
| `allow_parallel` | Позволяет Polars обрабатывать датафреймы вплоть до объединения в параллельном режиме. Значение по умолчанию – `True` |
| `force_parallel` | Вынуждает Polars обрабатывать датафреймы вплоть до объединения в параллельном режиме. Значение по умолчанию – `False` |

Перед рассмотрением примеров вам необходимо усвоить, что метод `join_asof()` будет корректно работать только при условии упорядочивания обоих датафреймов по объединяемым столбцам.

Давайте создадим два следующих датафрейма:

```
df_left = pl.DataFrame({"int_id": [10, 5], "value": ["b", "a"]})

df_right = pl.DataFrame({"int_id": [4, 7, 12], "value": [1, 2, 3]})
```

Если эти датафреймы не отсортированы, при попытке их неточного объединения вы получите ошибку, как показано ниже:

```
df_left.join_asof(df_right, on="int_id", tolerance=3)
```

Вывод:

```
InvalidOperationError: argument in operation 'asof_join' is not sorted, please
sort the 'expr/series/column' first
```

Таким образом, вам предварительно необходимо вызвать для неупорядоченных датафреймов метод `sort("int_id")`:

```
df_left = df_left.sort("int_id")
df_right = df_right

df_left.join_asof(df_right, on="int_id")
```

Вывод:

```
shape: (2, 3)
┌────────┬───────┬─────────────┐
| int_id | value | value_right |
| ---    | ---   | ---         |
```

```
| i64    | str    | i64          |
╞════════╪════════╪══════════════╡
| 5      | a      | 1            |
| 10     | b      | 2            |
└────────┴────────┴──────────────┘
```

Обратите внимание, что по умолчанию столбец объединения из правого датафрейма не включается в итоговую выборку, если его имя совпадает со столбцом объединения в левом датафрейме. Если это не то, что вам нужно, вы можете передать аргументу `coalesce` значение `False`, как показано ниже:

```
df_left.join_asof(
    df_right,
    on="int_id",
    coalesce=False,
)
```

Вывод:

```
shape: (2, 4)
┌────────┬────────┬──────────────┬──────────────┐
| int_id | value  | int_id_right | value_right  |
| ---    | ---    | ---          | ---          |
| i64    | str    | i64          | i64          |
╞════════╪════════╪══════════════╪══════════════╡
| 5      | a      | 4            | 1            |
| 10     | b      | 7            | 2            |
└────────┴────────┴──────────────┴──────────────┘
```

Если столбцы, по которым объединяются данные, называются по-разному, вы можете воспользоваться аргументами `left_on` и `right_on` для указания нужных столбцов для связывания:

```
df_left.join_asof(
    df_right.rename({"int_id": "int_id_right"}),
    left_on="int_id",
    right_on="int_id_right",
)
```

Вывод:

```
shape: (2, 4)
┌────────┬────────┬──────────────┬──────────────┐
| int_id | value  | int_id_right | value_right  |
| ---    | ---    | ---          | ---          |
| i64    | str    | i64          | i64          |
╞════════╪════════╪══════════════╪══════════════╡
| 5      | a      | 4            | 1            |
| 10     | b      | 7            | 2            |
└────────┴────────┴──────────────┴──────────────┘
```

# Стратегии неточного объединения

В методе `join_asof()` можно реализовать три стратегии неточного объединения, перечисленные ниже. Они передаются в текстовом виде аргументу `strategy`:

- `"backward"` (по умолчанию): объединение с последней строкой, в которой присутствует равное искомому или *меньшее* значение;
- `"forward"`: объединение с первой строкой, в которой присутствует равное искомому или *большее* значение;
- `"nearest"`: объединение со строкой, в которой присутствует ближайшее к искомому значение.

Давайте еще раз взглянем на датафреймы, которые помогут нам опробовать эти стратегии в деле:

```
print(df_left)
print(df_right)
```

Вывод:

```
shape: (2, 2)
┌────────┬───────┐
│ int_id ┆ value │
│ ---    ┆ ---   │
│ i64    ┆ str   │
╞════════╪═══════╡
│ 5      ┆ a     │
│ 10     ┆ b     │
└────────┴───────┘
shape: (3, 2)
┌────────┬───────┐
│ int_id ┆ value │
│ ---    ┆ ---   │
│ i64    ┆ i64   │
╞════════╪═══════╡
│ 4      ┆ 1     │
│ 7      ┆ 2     │
│ 12     ┆ 3     │
└────────┴───────┘
```

По умолчанию при нечетком объединении датафреймов используется стратегия `backward`. При объединении она ищет последнее значение во втором датафрейме, равное или меньшее искомого, но при этом не выходящее за границы, заданные при помощи аргумента `tolerance`:

```
df_left.join_asof(
    df_right,
    on="int_id",
```

```
    tolerance=3,
    strategy="backward",
)
```

Вывод:

```
shape: (2, 3)
┌────────┬───────┬─────────────┐
│ int_id ┆ value ┆ value_right │
│ ---    ┆ ---   ┆ ---         │
│ i64    ┆ str   ┆ i64         │
╞════════╪═══════╪═════════════╡
│ 5      ┆ a     ┆ 1           │
│ 10     ┆ b     ┆ 2           │
└────────┴───────┴─────────────┘
```

Стратегия `forward`, напротив, осуществляет поиск первого значения в присоединяемом датафрейме, равное или большее искомого, что видно ниже:

```
df_left.join_asof(
    df_right,
    on="int_id",
    tolerance=3,
    strategy="forward",
)
```

Вывод:

```
shape: (2, 3)
┌────────┬───────┬─────────────┐
│ int_id ┆ value ┆ value_right │
│ ---    ┆ ---   ┆ ---         │
│ i64    ┆ str   ┆ i64         │
╞════════╪═══════╪═════════════╡
│ 5      ┆ a     ┆ 2           │
│ 10     ┆ b     ┆ 3           │
└────────┴───────┴─────────────┘
```

Наконец, использование стратегии `nearest` поможет обнаружить наиболее близкие значения в правом датафрейме:

```
df_left.join_asof(
    df_right,
    on="int_id",
    tolerance=3,
    strategy="nearest",
)
```

Вывод:

```
shape: (2, 3)
┌────────┬───────┬─────────────┐
│ int_id ┆ value ┆ value_right │
```

```
| ---    | ---   | ---          |
| i64    | str   | i64          |
╞════════╪═══════╪══════════════╡
| 5      | a     | 1            |
| 10     | b     | 3            |
└────────┴───────┴──────────────┘
```

## Дополнительная тонкая настройка

При указании аргумента `tolerance` строки объединяются только в случае вхождения значения ключа из правой таблицы в заданный диапазон. В этом аргументе вы можете использовать как числовые, так и календарные значения. Для работы с датами и временем можно воспользоваться типом данных `datetime.timedelta` или строковыми представлениями длительности, которые мы перечисляли в табл. 13.5, например выражением `"7d12h30m"`.

Если вы хотите гарантировать, чтобы сначала объединение выполнялось по точному совпадению, а не по ближайшим значениям, вы можете воспользоваться аргументом `by`. И опять же, если в разных датафреймах ключевые столбцы именуются по-разному, вам помогут аргументы `by_left` и `by_right`.

Теперь давайте рассмотрим пример на объединение данных.

## Пример: управление маркетинговыми кампаниями

Представьте, что вам необходимо оценить эффективность ваших маркетинговых кампаний. Вы получили два набора данных, в одном из которых содержится информация о продажах, а во втором – маркетинговые кампании за последний год:

```
campaigns = pl.scan_csv("data/campaigns.csv")
campaigns.head(1).collect()
```

Вывод:

```
shape: (1, 3)
┌─────────────────┬─────────────────────┬──────────────┐
│ Campaign Name   ┆ Campaign Date       ┆ Product Type │
│ ---             ┆ ---                 ┆ ---          │
│ str             ┆ str                 ┆ str          │
╞═════════════════╪═════════════════════╪══════════════╡
│ Launch          ┆ 2023-01-01 20:00:00 ┆ Electronics  │
└─────────────────┴─────────────────────┴──────────────┘
```

```
campaigns.select(pl.col("Product Type").unique()).collect()
```

Вывод:

```
shape: (4, 1)
┌──────────────┐
│ Product Type │
│ ---          │
│ str          │
╞══════════════╡
│ Electronics  │
│ Furniture    │
│ Clothing     │
│ Books        │
└──────────────┘
```

```
transactions = pl.scan_csv("data/transactions.csv")
transactions.head(1).collect()
```

Вывод:

```
shape: (1, 3)
┌──────────────────────┬──────────────┬──────────┐
│ Sale Date            │ Product Type │ Quantity │
│ ---                  │ ---          │ ---      │
│ str                  │ str          │ i64      │
╞══════════════════════╪══════════════╪══════════╡
│ 2023-01-01 02:00:00…. │ Books        │ 7        │
└──────────────────────┴──────────────┴──────────┘
```

Как видите, все даты у нас пока представлены с типом данных `String`. Для работы с такими данными вам сначала нужно привести их к соответствующему типу. К тому же, поскольку значения времени у нас могут в точности не совпадать, придется воспользоваться методом `join_asof()` для объединения двух датафреймов. Дополнительно нам необходимо сопоставить маркетинговые кампании с категориями продаваемых товаров, что можно сделать при помощи аргумента `by`. И последнее замечание: кампании не длятся вечно, так что передадим аргументу `tolerance` значение, соответствующее двум месяцам:

```
transactions = transactions.with_columns(
    pl.col("Sale Date")
    .str.to_datetime("%Y-%m-%d %H:%M:%S%.f")
    .cast(pl.Datetime("us")),
)
campaigns = campaigns.with_columns(
    pl.col("Campaign Date").str.to_datetime("%Y-%m-%d %H:%M:%S"),
)

sales_with_campaign_df = (
    transactions.sort("Sale Date")
    .join_asof(
        campaigns.sort("Campaign Date"),
```

```
        left_on="Sale Date",
        right_on="Campaign Date",
        by="Product Type",
        strategy="backward",
        tolerance="60d",
    )
    .collect()
)
sales_with_campaign_df
```

Вывод:

```
shape: (20_000, 5)
```

| Sale Date | Product Type | Quantity | Campaign Name | Campaign Date |
|---|---|---|---|---|
| datetime[μs] | str | i64 | str | datetime[μs] |
| 2023-01-01 01:26:12…. | Electronics | 2 | null | null |
| 2023-01-01 02:00:00 | Books | 7 | null | null |
| 2023-01-01 06:14:30…. | Toys | 9 | null | null |
| 2023-01-01 06:52:25…. | Clothing | 9 | null | null |
| 2023-01-01 07:44:50…. | Books | 7 | null | null |
| … | … | … | … | … |
| 2023-12-31 15:45:29…. | Clothing | 10 | null | null |
| 2023-12-31 18:15:09…. | Toys | 4 | null | null |
| 2023-12-31 18:33:47…. | Electronics | 7 | null | null |
| 2023-12-31 18:37:54…. | Books | 6 | null | null |
| 2023-12-31 19:41:22…. | Furniture | 4 | null | null |

Теперь если вы захотите понять, привели ли маркетинговые кампании к повышению средних продаж, то можете сгруппировать данные по столбцам `Product Type` и `Campaign Name`. Это позволит сравнить средние продажи одних и тех же категорий товаров в рамках маркетинговой кампании и вне кампании, как показано ниже:

```
(
    sales_with_campaign_df.group_by("Product Type", "Campaign Name")
    .agg(pl.col("Quantity").mean())
    .sort("Product Type", "Campaign Name")
)
```

Вывод:

```
shape: (9, 3)
┌──────────────┬─────────────────┬──────────┐
│ Product Type ┆ Campaign Name   ┆ Quantity │
│ ---          ┆ ---             ┆ ---      │
│ str          ┆ str             ┆ f64      │
╞══════════════╪═════════════════╪══════════╡
│ Books        ┆ null            ┆ 5.527716 │
│ Clothing     ┆ null            ┆ 5.433385 │
│ Clothing     ┆ New Arrivals    ┆ 8.200581 │
│ Electronics  ┆ null            ┆ 5.486832 │
│ Electronics  ┆ Launch          ┆ 8.080775 │
│ Electronics  ┆ Seasonal Sale   ┆ 8.471406 │
│ Furniture    ┆ null            ┆ 5.430222 │
│ Furniture    ┆ Discount        ┆ 8.191888 │
│ Toys         ┆ null            ┆ 5.50318  │
└──────────────┴─────────────────┴──────────┘
```

Анализируя полученные результаты, можно прийти к выводу, что участие в маркетинговых кампаниях обычно приводит к увеличению средних продаж. Данные о продажах в рамках кампаний отсутствуют только по категориям Books и Toys. Посмотрим, что с ними не так.

В категории Toys у нас не было ни одной рекламной кампании, а что насчет категории Books? Узнаем:

```
campaigns.filter(pl.col("Product Type") == "Books").collect()
```

Вывод:

```
shape: (1, 3)
┌───────────────┬─────────────────────┬──────────────┐
│ Campaign Name ┆ Campaign Date       ┆ Product Type │
│ ---           ┆ ---                 ┆ ---          │
│ str           ┆ datetime[μs]        ┆ str          │
╞═══════════════╪═════════════════════╪══════════════╡
│ Clearance     ┆ 2023-12-31 21:00:00 ┆ Books        │
└───────────────┴─────────────────────┴──────────────┘
```

Похоже, в этой категории у нас была только одна рекламная акция в виде рождественской распродажи. Давайте узнаем, были ли продажи книг после даты начала распродажи:

```
(
    transactions.filter(
        (pl.col("Product Type") == "Books")
        & (
            pl.col("Sale Date")
            > pl.lit("2023-12-31 21:00:00").str.to_datetime()
        )
    ).collect()
)
```

Вывод:

```
shape: (0, 3)
┌──────────────┬──────────────┬──────────┐
│ Sale Date    ┆ Product Type ┆ Quantity │
│ ---          ┆ ---          ┆ ---      │
│ datetime[μs] ┆ str          ┆ i64      │
╞══════════════╪══════════════╪══════════╡
└──────────────┴──────────────┴──────────┘
```

Как видите, ни одной книги во время распродажи мы не продали. А поскольку в методе `join_asof()` мы использовали стратегию `"backward"`, кампания не была присвоена ни одной продаже из категории Books, что объясняет полученный нами результат.

**Неэквивалентные объединения**

На момент написания книги в Polars уже присутствовала поддержка так называемых *неэквивалентных объединений* (non-equi join). Эти объединения, реализованные при помощи метода *`join_where()`*, используются для связи между датафреймами на основе неравенств. Поскольку в данный момент эти объединения обладают экспериментальным статусом, мы не будем подробно рассказывать о них в книге. Но знать о них нужно.

# Вертикальное и горизонтальное слияние

Метод `join()`, о котором мы подробно поговорили в предыдущем разделе, служит для объединения датафреймов на основе присутствующих в них значений. Но иногда вам требуется просто присоединить один датафрейм к другому, не обращая внимания на значения в них. Обычно разные датафреймы хранятся в разных участках памяти. При их *слиянии* (concatenation) вы можете пойти тремя путями:

- объединить все данные в отдельный датафрейм путем копирования его в новое место в памяти;
- указать новому датафрейму место размещения данных;
- скопировать данные из второго датафрейма и разместить их непосредственно после данных из первого.

Первый способ связан с копированием данных в новое место. Это поведение по умолчанию для функции *`pl.concat()`*, принимающей список датафреймов, ленивых датафреймов или объектов Series и выполняющей их слияние по горизонтали, вертикали или диагонали. После слияния данных производится разбиение набора на блоки и копирование в новое место. Это обеспечивает эффективную работу с новым датафреймом в будущем.

Как мы узнаем в главе 18, при разбиении на блоки данные копируются в новое место, образуя непрерывный сегмент в памяти. Это весьма полезно

в плане эффективности запросов к новому датафрейму в дальнейшем, особенно если эти запросы будут использоваться многократно.

В табл. 14.3 перечислены аргументы, которые может принимать функция `pl.concat()`.

***Таблица 14.3. Аргументы функции*** *`pl.concat()`*

| Аргумент | Описание |
|---|---|
| `Items` | Список датафреймов, ленивых датафреймов или объектов Series для слияния |
| `How` | Стратегия слияния |
| `Rechunk` | Указывает на необходимость разбиения на блоки итогового датафрейма. Значение по умолчанию – `False` |
| `Parallel` | Определяет, могут ли ленивые датафреймы обрабатываться параллельно. Значение по умолчанию – `True` |

Аргумент `how` может принимать следующие строковые значения, соответствующие используемой стратегии слияния: `"vertical"`, `"vertical_relaxed"`, `"horizontal"`, `"diagonal"`, `"diagonal_relaxed"` и `"align"`[1].

# Вертикальное слияние

Первым делом рассмотрим операцию *вертикального слияния* (vertical concatenation). Эта стратегия используется в методе `pl.concat()` по умолчанию и заключается в объединении датафреймов по вертикали, т. е. в присоединении строк из разных датафреймов друг к другу. На рис. 14.1 схематически показано, как выполняется вертикальное слияние.

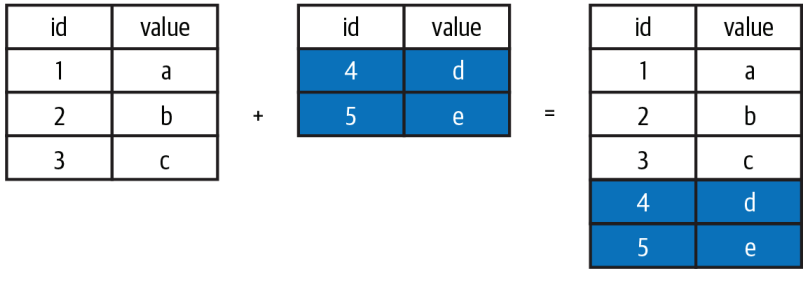



**Рис. 14.1** ❖ Вертикальное слияние

Давайте рассмотрим пример вертикального слияния на практике:

```
df1 = pl.DataFrame(
    {
```

[1] Впоследствии были добавлены стратегии `"align_full"`, `"align_left"` и `"align_right"`. – *Прим. перев.*

```
        "id": [1, 2, 3],
        "value": ["a", "b", "c"],
    }
)
df2 = pl.DataFrame(
    {
        "id": [4, 5],
        "value": ["d", "e"],
    }
)
pl.concat([df1, df2], how="vertical")
```

Вывод:

```
shape: (5, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ i64 ┆ str   │
╞═════╪═══════╡
│ 1   ┆ a     │
│ 2   ┆ b     │
│ 3   ┆ c     │
│ 4   ┆ d     │
│ 5   ┆ e     │
└─────┴───────┘
```

## Горизонтальное слияние

Вторая стратегия называется *горизонтальным слиянием* (horizontal concatenation). При ее использовании датафреймы объединяются по горизонтали, т. е. их столбцы размещаются рядом друг с другом. При несоответствии размеров исходных датафреймов оставшиеся значения будут заполнены значениями `null`. При этом датафреймы не могут содержать столбцы с одинаковыми именами – в этом случае вернется ошибка. Избежать этого можно путем предварительного переименования столбцов. На рис. 14.2 схематически показано, как выполняется горизонтальное слияние.

| id | value |
|---|---|
| 1 | a |
| 2 | b |
| 3 | c |

+

| value2 |
|---|
| x |
| y |

=

| id | value | value2 |
|---|---|---|
| 1 | a | x |
| 2 | b | y |
| 3 | c | null |

**Рис. 14.2** ❖ Горизонтальное слияние

Рассмотрим пример:

```
df1 = pl.DataFrame(
    {
        "id": [1, 2, 3],
        "value": ["a", "b", "c"],
    }
)
df2 = pl.DataFrame(
    {
        "value2": ["x", "y"],
    }
)
pl.concat([df1, df2], how="horizontal")
```

Вывод:

```
shape: (3, 3)
┌─────┬───────┬────────┐
│ id  ┆ value ┆ value2 │
│ --- ┆ ---   ┆ ---    │
│ i64 ┆ str   ┆ str    │
╞═════╪═══════╪════════╡
│ 1   ┆ a     ┆ x      │
│ 2   ┆ b     ┆ y      │
│ 3   ┆ c     ┆ null   │
└─────┴───────┴────────┘
```

# Диагональное слияние

Третья стратегия слияния осуществляется *по диагонали* (diagonal concatenation). Это слияние выполняется путем объединения столбцов исходных датафреймов. Все недостающие значения при этом заменяются на значения `null`. На рис. 14.3 показана схема выполнения диагонального слияния.

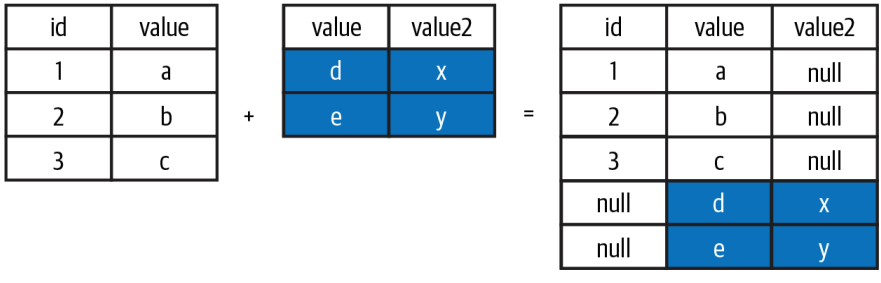



**Рис. 14.3** ❖ Диагональное слияние

Рассмотрим пример:

```
df1 = pl.DataFrame(
    {
        "id": [1, 2, 3],
        "value": ["a", "b", "c"],
    }
)
df2 = pl.DataFrame(
    {
        "value": ["d", "e"],
        "value2": ["x", "y"],
    }
)
pl.concat([df1, df2], how="diagonal")
```

Вывод:

```
shape: (5, 3)
┌──────┬───────┬────────┐
│ id   ┆ value ┆ value2 │
│ ---  ┆ ---   ┆ ---    │
│ i64  ┆ str   ┆ str    │
╞══════╪═══════╪════════╡
│ 1    ┆ a     ┆ null   │
│ 2    ┆ b     ┆ null   │
│ 3    ┆ c     ┆ null   │
│ null ┆ d     ┆ x      │
│ null ┆ e     ┆ y      │
└──────┴───────┴────────┘
```

## Объединяющее слияние

Четвертым, и последним, типом слияния датафреймов в Polars является так называемое *объединяющее слияние* (align concatenation). При использовании этой стратегии строки и столбцы датафреймов не просто присоединяются друг к другу, а выполняется поиск совпадений по столбцам, и строки размещаются в итоговом датафрейме в соответствии с найденными совпадениями. Отсутствие совпадений приводит к образованию значений `null`. Схематически этот принцип показан на рис. 14.4.

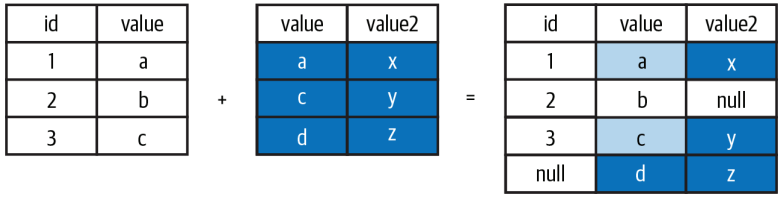



**Рис. 14.4** ❖ Объединяющее слияние

Рассмотрим пример объединяющего слияния:

```
df1 = pl.DataFrame(
    {
        "id": [1, 2, 3],
        "value": ["a", "b", "c"],
    }
)
df2 = pl.DataFrame(
    {
        "value": ["a", "c", "d"],
        "value2": ["x", "y", "z"],
    }
)
pl.concat([df1, df2], how="align")
```

Вывод:

```
shape: (4, 3)
┌──────┬───────┬────────┐
│ id   ┆ value ┆ value2 │
│ ---  ┆ ---   ┆ ---    │
│ i64  ┆ str   ┆ str    │
╞══════╪═══════╪════════╡
│ 1    ┆ a     ┆ x      │
│ 2    ┆ b     ┆ null   │
│ 3    ┆ c     ┆ y      │
│ null ┆ d     ┆ z      │
└──────┴───────┴────────┘
```

Операция объединяющего слияния базируется на функции *pl.align_frames()*. Эта функция позволяет выбрать столбец и расположить строки в наборе датафреймов в соответствии со значениями в этом столбце. Если в одном из датафреймов значение будет отсутствовать, в итоговую строку результирующего датафрейма будет вставлено значение `null`. Если значения встречаются несколько раз, будет создано декартово произведение строк. В результате функция возвращает те же датафреймы, но с добавленными строками из них же самих в соответствии с найденными совпадениями. Лучше это будет понятно на примере, показанном ниже. В табл. 14.4 перечислены аргументы, которые может принимать функция `pl.align_frames()`.

***Таблица 14.4. Аргументы функции*** *pl.align_frames()*

| Аргумент | Описание |
|---|---|
| `*frames` | Датафреймы для слияния |
| `*on` | Столбец, по которому будет осуществляться поиск соответствий |
| `How` | Стратегия слияния, определяющая вид результирующих датафреймов. Значение по умолчанию – `full` |
| `Select` | Столбцы и их порядок для выбора из итоговых датафреймов |
| `descending` | Указывает, нужно ли сортировать по убыванию итоговые датафреймы. Также может принимать список булевых значений с количеством элементов, соответствующим длине столбца из аргумента `on` |

Следующий фрагмент кода демонстрирует работу функции `pl.align_frames()`:

```
df1 = pl.DataFrame(
    {
        "id": [1, 2, 2],
        "value": ["a", "c", "b"],
    }
)
df2 = pl.DataFrame(
    {
        "id": [2, 2],
        "value": ["x", "y"],
    }
)
pl.align_frames(df1, df2, on="id")
```

Вывод:

```
[shape: (5, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ i64 ┆ str   │
╞═════╪═══════╡
│ 1   ┆ a     │
│ 2   ┆ c     │
│ 2   ┆ b     │
│ 2   ┆ c     │
│ 2   ┆ b     │
└─────┴───────┘,
shape: (5, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ i64 ┆ str   │
╞═════╪═══════╡
│ 1   ┆ null  │
│ 2   ┆ x     │
│ 2   ┆ x     │
│ 2   ┆ y     │
│ 2   ┆ y     │
└─────┴───────┘]
```

Обратите внимание, что во втором датафрейме, в котором отсутствовал идентификатор 1, соответствующая строка оказалась заполнена значением `null`. Кроме того, поскольку в исходных датафреймах было по две строки с идентификатором 2, в результирующих датафреймах появились декартовы произведения исходных строк с этим идентификатором.

# Нестрогие виды слияния

Стратегии вертикального и диагонального слияния датафреймов также располагают своими нестрогими версиями. Если при их использовании будет обнаружено, что типы данных в столбцах с одинаковыми именами не совпадают, эти столбцы будут автоматически приведены к *супертипу* (supertype). К примеру, столбцы с целочисленным и вещественным типами данных будут приведены к вещественному типу, а столбцы с целочисленным и строковым типами – к строковому. Это бывает полезно при необходимости выполнить слияние датафреймов с одинаковыми столбцами, но отличающимися типами данных. Взгляните на следующий пример:

```
df1 = pl.DataFrame(
    {
        "id": [1, 2, 3],
        "value": ["a", "b", "c"],
    }
)
df2 = pl.DataFrame(
    {
        "id": [4.0, 5.0],
        "value": [1, 2],
    }
)
pl.concat([df1, df2], how="vertical")
```

Вывод:

```
SchemaError: type Float64 is incompatible with expected type Int64
```

Все правильно, мы получили ошибку, связанную с несоответствием типов данных. Однако при использовании стратегии `"vertical_relaxed"` слияние будет выполнено успешно, как показано ниже:

```
pl.concat([df1, df2], how="vertical_relaxed")
```

Вывод:

```
shape: (5, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ f64 ┆ str   │
╞═════╪═══════╡
│ 1.0 ┆ a     │
│ 2.0 ┆ b     │
│ 3.0 ┆ c     │
│ 4.0 ┆ 1     │
│ 5.0 ┆ 2     │
└─────┴───────┘
```

# Стекинг

Методы датафреймов *vstack()* и *hstack()* предназначены для выполнения еще одного вида слияния датафреймов, а именно *стекинга* (stacking). Для объектов Series существует схожий метод с именем `Series.append()`. Эти методы позволяют объединять датафреймы без перемещения данных в памяти. Вместо этого создается новый датафрейм или объект Series, содержащий множество блоков, которые могут размещаться в разных участках памяти. Это позволяет значительно ускорить выполнение операции слияния, а в жертву приносится скорость выполнения запросов к итоговому объекту из-за необходимости обращаться к разным участкам памяти. Стратегия `"vertical"` в функции `pl.concat()` как раз использует метод `vstack()`, но его можно вызывать и отдельно. При этом функция `pl.concat()` позволяет разбить результирующий датафрейм на блоки, предотвращая жертвование скоростью, тогда как метод `vstack()` такой возможности не имеет.

Эти методы предпочтительно использовать при непосредственном добавлении датафреймов друг к другу. Обратите внимание, что операции стекинга могут выполняться только применительно к традиционным датафреймам, а не ленивым, поскольку они объединяют существующие блоки.

Метод `vstack()` требует совпадения количества столбцов, их имен и типов данных в объединяемых датафреймах:

```
df1 = pl.DataFrame(
    {
        "id": [1, 2],
        "value": ["a", "b"],
    }
)
df2 = pl.DataFrame(
    {
        "id": [3, 4],
        "value": ["c", "d"],
    }
)
df1.vstack(df2)
```

Вывод:

```
shape: (4, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ i64 ┆ str   │
╞═════╪═══════╡
│ 1   ┆ a     │
│ 2   ┆ b     │
│ 3   ┆ c     │
│ 4   ┆ d     │
└─────┴───────┘
```

Подобно тому как метод `vstack()` объединяет датафреймы по вертикали, метод `hstack()` делает то же самое по горизонтали. Эта операция требует совпадения количества строк в обоих датафреймах:

```
df1 = pl.DataFrame(
    {
        "id": [1, 2],
        "value": ["a", "b"],
    }
)
df2 = pl.DataFrame(
    {
        "value2": ["x", "y"],
    }
)
df1.hstack(df2)
```

Вывод:

```
shape: (2, 3)
┌─────┬───────┬────────┐
│ id  ┆ value ┆ value2 │
│ --- ┆ ---   ┆ ---    │
│ i64 ┆ str   ┆ str    │
╞═════╪═══════╪════════╡
│ 1   ┆ a     ┆ x      │
│ 2   ┆ b     ┆ y      │
└─────┴───────┴────────┘
```

# Добавление

Для *добавления* (append) одного объекта Series к другому необходимо воспользоваться методом *append()* объекта Series. В результате выполнения операции будет сохранено имя исходного объекта Series, что показано ниже:

```
series_a = pl.Series("a", [1, 2])
series_b = pl.Series("b", [3, 4])
series_a.append(series_b)
```

Вывод:

```
shape: (4,)
Series: 'a' [i64]
[
    1
    2
    3
    4
]
```

# Расширение

Еще один способ комбинирования датафреймов состоит в их *расширении* (extending) с помощью метода *extend()*. Если в памяти достаточно места для размещения нового датафрейма по соседству с исходным, метод `extend()` скопирует данные из второго датафрейма и разместит их следом за первым. Это позволяет избежать копирования всех данных в новое место, что может сэкономить время на выполнение операции. В то же время мы располагаем данные в памяти последовательно, что сокращает время на выполнение последующих запросов. Этот способ хорошо подходит, когда необходимо добавить небольшой датафрейм к довольно объемному. Пример показан ниже:

```
df1 = pl.DataFrame(
    {
        "id": [1, 2],
        "value": ["a", "b"],
    }
)
df2 = pl.DataFrame(
    {
        "id": [3, 4],
        "value": ["c", "d"],
    }
)
df1.extend(df2)
```

Вывод:

```
shape: (4, 2)
┌─────┬───────┐
│ id  ┆ value │
│ --- ┆ ---   │
│ i64 ┆ str   │
╞═════╪═══════╡
│ 1   ┆ a     │
│ 2   ┆ b     │
│ 3   ┆ c     │
│ 4   ┆ d     │
└─────┴───────┘
```

**На месте стой!**

Метод `extend()` – один из немногих в Polars, который выполняется на месте. Это означает, что в результате его вызова модифицируется исходный датафрейм, а не создается его копия. При этом данный метод возвращает измененный датафрейм, но это сделано исключительно ради удобства.

# Заключение

В этой главе вы научились объединять и комбинировать датафреймы в Polars. Вы узнали, как можно:

- объединять датафреймы с использованием точных соответствий при помощи метода `join()`. Этот метод располагает множеством параметров, позволяющих тонко настроить стратегию объединения;
- объединять датафреймы по числовым и календарным столбцам с применением неточных соответствий с помощью метода `join_asof()`;
- добавлять датафреймы друг к другу по вертикали при наличии совпадающих столбцов при помощи функции `pl.concat()` и метода `vstack()`;
- комбинировать датафреймы по горизонтали при помощи функции `pl.concat()` и метода `hstack()`;
- добавлять строки датафрейма посредством метода `extend()`;
- добавлять объекты Series друг к другу с помощью метода `series.append()`.

В следующей главе мы узнаем, как можно изменять форму датафреймов, что бывает полезно при модифицировании их структуры.

# Глава 15

# Изменение формы датафреймов

В одной из предыдущих глав мы говорили о группировке и агрегации данных для создания информативных агрегаций. Но что делать, если исходные данные не обладают формой, приемлемой для расчета нужных нам агрегаций? Изменение формы датасетов является одним из ключевых шагов в процессе анализа данных.

В этой главе вы узнаете, как:

- изменять форму данных для их большей пригодности для анализа;
- вносить изменения в размерности данных с целью повышения эффективности их обработки или подготовки к визуализации;
- применять полезные методы датафреймов в Polars, такие как `pivot()`, `unpivot()`, `transpose()`, `explode()` и `partition_by()`.

Все инструкции по загрузке нужных файлов и установке пакетов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

## Широкие датафреймы против длинных

*Широкие датафреймы* (wide dataframe) характеризуются большим количеством столбцов и небольшим количеством строк. Идея таких датафреймов состоит в том, что они содержат одну колонку с идентификатором и множество столбцов с самими данными, характеризующими тот или иной объект, или наблюдение. Такой формат хранения данных используется в основном тогда, когда одно наблюдение описывается большим количеством измерений, или параметров. Пример широкого датафрейма представлен ниже:

```
grades_wide = pl.DataFrame(
    {
        "student": ["Jeroen", "Thijs", "Ritchie"],
```

```
        "math": [85, 78, 92],
        "science": [90, 82, 85],
        "history": [88, 80, 87],
    }
)

grades_wide
```

В этом датафрейме по успеваемости студентов оценки за конкретные предметы хранятся в отдельных столбцах, имена которых соответствуют названиям дисциплин.

В противоположность широким датафреймам *длинные* (long dataframe) не могут похвастаться большим количеством столбцов, зато содержат множество строк. В нашем примере, вместо того чтобы хранить оценки по предметам в отдельных столбцах, мы могли бы создать один столбец с названиями дисциплин и один – с оценками. Тогда наш предыдущий пример мог бы выглядеть так:

```
grades_long = pl.DataFrame(
    {
        "student": [
            "Jeroen",
            "Jeroen",
            "Jeroen",
            "Thijs",
            "Thijs",
            "Thijs",
            "Ritchie",
            "Ritchie",
            "Ritchie",
        ],
        "subject": [
            "Math",
            "Science",
            "History",
            "Math",
            "Science",
            "History",
            "Math",
            "Science",
            "History",
        ],
        "grade": [85, 90, 88, 78, 82, 80, 92, 85, 87],
    }
)

grades_long
```

Вывод:

```
shape: (9, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
```

```
| ---     | ---     | ---  |
| str     | str     | i64  |
╞═════════╪═════════╪══════╡
| Jeroen  | Math    | 85   |
| Jeroen  | Science | 90   |
| Jeroen  | History | 88   |
| Thijs   | Math    | 78   |
| Thijs   | Science | 82   |
| Thijs   | History | 80   |
| Ritchie | Math    | 92   |
| Ritchie | Science | 85   |
| Ritchie | History | 87   |
└─────────┴─────────┴──────┘
```

В таком виде в датафрейме каждое измерение (оценка по конкретному предмету) содержится в отдельной строке.

От формата хранения данных в датафреймах напрямую зависит эффективность работы с памятью и выполнения операций над данными. Поскольку в Polars используется колоночный тип хранения, длинные датафреймы обрабатываются в нем гораздо более эффективно.

### Опрятные данные

В своей знаменитой статье 2014 года под названием «Tidy Data» Хэдли Уикем (Hadley Wickham) описал разные форматы хранения данных. Хэдли – очень известная личность в сообществе, а свой основной вклад он внес в разработку и популяризацию языка программирования R. В частности, он является автором набора пакетов `tidyverse`, ставшего краеугольным камнем в деле обработки и визуализации данных в R.

В этой статье Хэдли впервые ввел концепцию широких и длинных датафреймов. Эти термины очень быстро закрепились в сообществе и сегодня повсеместно, в том числе и в этой книге, используются для описания форматов хранения табличных данных.

В дополнение к открытию терминологии для формы данных Хэдли Уикем также ввел в оборот термин *tidy data*, который можно перевести как опрятные, или аккуратные, данные.

По утверждению Уикема, набор данных должен отвечать трем следующим правилам, чтобы он мог именоваться опрятным, или аккуратным.

1. Каждой переменной должен быть выделен свой столбец.
2. Каждому наблюдению должна отводиться одна строка.
3. Каждое значение должно находиться в отдельной ячейке.

На рис. 15.1 эти правила показаны схематически.

В нашем примере с успеваемостью студентов широкий формат никак нельзя назвать опрятным. Каждый предмет в нем располагается в отдельном столбце, хотя на самом деле названия предметов (`science`, `history` и `math`) сами являются значениями переменной, характеризующей дисциплину. В длинном формате этот недостаток отсутствует, и именно это позволяет говорить о том, что данный формат отвечает всем трем правилам опрятности данных.


**Рис. 15.1** ❖ Пример набора данных, отвечающего всем трем правилам опрятности данных

# Разворачивание в широкий формат

Если вам необходимо перейти от длинного формата данных к широкому, вы можете воспользоваться методом датафрейма *pivot()*, принимающим показанные в табл. 15.1 аргументы.

***Таблица 15.1. Аргументы метода*** *pivot()*

| Аргумент | Описание |
|---|---|
| Index | Столбцы, данные из которых будут использоваться как идентификаторы для строк |
| Columns | Столбцы, содержащие имена колонок, которые будут созданы в результате выполнения операции |
| Values | Столбцы, содержащие значения, которые будут размещены в соответствующих ячейках |
| aggregate_function | Функция агрегации, которая будет применена в случае наличия нескольких значений для одной ячейки. Если оставить этот аргумент незаполненным, при встрече с несколькими значениями, которые должны перекочевать в одну ячейку, возникнет ошибка |
| maintain_order | Сортирует сгруппированные ключи для более предсказуемого вывода |
| sort_columns | Сортирует транспонированные столбцы по имени. По умолчанию сортировка выполняется в соответствии с порядком появления значений в исходном датафрейме |
| Separator | Разделитель для сгенерированных имен столбцов |

Давайте посмотрим на примере. Нам нужно хранить оценки группы студентов. Поскольку оценки поступают одна за одной, мы будем хранить данные в длинном формате, как показано ниже:

```
grades = pl.DataFrame(
    {
        "student": [
            "Jeroen",
```

```
            "Jeroen",
            "Jeroen",
            "Thijs",
            "Thijs",
            "Thijs",
            "Ritchie",
            "Ritchie",
            "Ritchie",
        ],
        "subject": [
            "Math",
            "Science",
            "History",
            "Math",
            "Science",
            "History",
            "Math",
            "Science",
            "History",
        ],
        "grade": [85, 90, 88, 78, 82, 80, 92, 85, 87],
    }
)

grades
```

Вывод:

```
shape: (9, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
│ ---     ┆ ---     ┆ ---   │
│ str     ┆ str     ┆ i64   │
╞═════════╪═════════╪═══════╡
│ Jeroen  ┆ Math    ┆ 85    │
│ Jeroen  ┆ Science ┆ 90    │
│ Jeroen  ┆ History ┆ 88    │
│ Thijs   ┆ Math    ┆ 78    │
│ Thijs   ┆ Science ┆ 82    │
│ Thijs   ┆ History ┆ 80    │
│ Ritchie ┆ Math    ┆ 92    │
│ Ritchie ┆ Science ┆ 85    │
│ Ritchie ┆ History ┆ 87    │
└─────────┴─────────┴───────┘
```

Довольно опрятно. После получения всех оценок студентов нам необходимо преобразовать данные так, чтобы каждому студенту соответствовала одна строка. Этого можно добиться, развернув данные по столбцу `subject` следующим образом:

```
grades.pivot(index="student", on="subject", values="grade")
```

Вывод:

```
shape: (3, 4)
┌─────────┬──────┬─────────┬─────────┐
│ student ┆ Math ┆ Science ┆ History │
│ ---     ┆ ---  ┆ ---     ┆ ---     │
│ str     ┆ i64  ┆ i64     ┆ i64     │
╞═════════╪══════╪═════════╪═════════╡
│ Jeroen  ┆ 85   ┆ 90      ┆ 88      │
│ Thijs   ┆ 78   ┆ 82      ┆ 80      │
│ Ritchie ┆ 92   ┆ 85      ┆ 87      │
└─────────┴──────┴─────────┴─────────┘
```

Здесь при переходе в широкий формат мы взяли имена студентов за ось разворота, а на основе дисциплин создали новые столбцы.

В реальности у каждого студента может быть не одна, а несколько оценок по каждому предмету. Это означает, что вам может понадобиться неким образом агрегировать их. К счастью, метод `pivot()` предусматривает такую возможность[1].

По умолчанию метод `pivot()` не агрегирует данные и вернет ошибку, если в одной ячейке окажется несколько значений. В нашем предыдущем примере этого не происходит, т. к. по сочетанию столбцов со студентом и предметом значения у нас были уникальны. Но если в вашем случае это не так и вам нужно как-то агрегировать значения, оказывающиеся в одной ячейке таблицы, вы можете передать методу `pivot` аргумент `aggregate_function` в виде строкового названия функции агрегации. На выбор предоставляются следующие функции: `min`, `max`, `first`, `last`, `sum`, `mean`, `median` и `len`. В нашем случае для расчета средних оценок мы можем воспользоваться функцией агрегации `mean`. Сначала создадим датафрейм со множеством оценок для студентов по одному и тому же предмету:

```
multiple_grades = pl.DataFrame(
    {
        "student": [
            "Jeroen",
            "Jeroen",
            "Jeroen",
            "Jeroen",
            "Jeroen",
            "Jeroen",
            "Thijs",
            "Thijs",
            "Thijs",
            "Thijs",
            "Thijs",
            "Thijs",
        ],
```

[1] Даже Росс из сериала «Друзья» понял бы, что выполнить команду «Поворот!» применительно к датафрейму проще, чем при затаскивании дивана вверх по угловой лестнице.

```
        "subject": [
            "Math",
            "Math",
            "Math",
            "Science",
            "Science",
            "Science",
            "Math",
            "Math",
            "Math",
            "Science",
            "Science",
            "Science",
        ],
        "grade": [85, 88, 85, 60, 66, 63, 51, 79, 62, 82, 85, 82],
    }
)

multiple_grades
```

Вывод:

```
shape: (12, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
│ ---     ┆ ---     ┆ ---   │
│ str     ┆ str     ┆ i64   │
╞═════════╪═════════╪═══════╡
│ Jeroen  ┆ Math    ┆ 85    │
│ Jeroen  ┆ Math    ┆ 88    │
│ Jeroen  ┆ Math    ┆ 85    │
│ Jeroen  ┆ Science ┆ 60    │
│ Jeroen  ┆ Science ┆ 66    │
│ …       ┆ …       ┆ …     │
│ Thijs   ┆ Math    ┆ 79    │
│ Thijs   ┆ Math    ┆ 62    │
│ Thijs   ┆ Science ┆ 82    │
│ Thijs   ┆ Science ┆ 85    │
│ Thijs   ┆ Science ┆ 82    │
└─────────┴─────────┴───────┘
```

Теперь мы можем развернуть наш датафрейм по студентам с подсчетом средних оценок для них следующим образом:

```
multiple_grades.pivot(
    index="student", on="subject", values="grade", aggregate_function="mean"
)
```

Вывод:

```
shape: (2, 3)
┌─────────┬──────┬─────────┐
│ student ┆ Math ┆ Science │
│ ---     ┆ ---  ┆ ---     │
```

```
| str     | f64  | f64     |
╞═════════╪══════╪═════════╡
| Jeroen  | 86.0 | 63.0    |
| Thijs   | 64.0 | 83.0    |
└─────────┴──────┴─────────┘
```

В дополнение к списку стандартных функций агрегации вы также можете передать пользовательскую функцию посредством выражений. Создав выражение, которое может быть применено к элементам списка (вроде следующего: `pl.col(...).list.eval(<ваше выражение>)`), вы можете использовать его для придания гибкости агрегациям. К примеру, вы можете вычислить разницу между максимальной и минимальной оценками для студентов, чтобы выяснить стабильность показываемых результатов, как видно ниже:

```
multiple_grades.pivot(
    index="student",
    on="subject",
    values="grade",
    aggregate_function=pl.element().max() - pl.element().min(),
)
```

Вывод:

```
shape: (2, 3)
┌─────────┬──────┬─────────┐
| student | Math | Science |
| ---     | ---  | ---     |
| str     | i64  | i64     |
╞═════════╪══════╪═════════╡
| Jeroen  | 3    | 6       |
| Thijs   | 28   | 3       |
└─────────┴──────┴─────────┘
```

Как видите, у Тейса гораздо большая разница между максимальной и минимальной оценками по математике в сравнении с Йеруном. Это могло бы стать поводом, чтобы удостовериться у преподавателя, есть ли у Тейса темы по математике, которые ему стоит подтянуть.

### Ленивый разворот

При выполнении разворачивания данных Polars необходимо знать, какие значения находятся в датафрейме, чтобы перенести их в новые столбцы. Это означает, что вы не можете применить метод `pivot()` к ленивому датафрейму напрямую.

Но есть обходной путь. Если вы заранее знаете, какие столбцы будут присутствовать в ленивом датафрейме, вы можете выполнить разворот следующим образом:

```
lf = pl.LazyFrame(
    {
        "col1": ["a", "a", "a", "b", "b", "b"],
        "col2": ["x", "x", "x", "x", "y", "y"],
```

```
        "col3": [6, 7, 3, 2, 5, 7],
    }
)

index = pl.col("col1")
on = pl.col("col2")
values = pl.col("col3")
unique_column_values = ["x", "y"]
aggregate_function = lambda col: col.tanh().mean()

lf.group_by(index).agg(
    aggregate_function(values.filter(on == value)).alias(value)
    for value in unique_column_values
).collect()
```

Вывод:

```
shape: (2, 3)
┌──────┬──────────┬──────────┐
│ col1 ┆ x        ┆ y        │
│ ---  ┆ ---      ┆ ---      │
│ str  ┆ f64      ┆ f64      │
╞══════╪══════════╪══════════╡
│ a    ┆ 0.998347 ┆ null     │
│ b    ┆ 0.964028 ┆ 0.999954 │
└──────┴──────────┴──────────┘
```

# Разворачивание в длинный формат

Если вы, наоборот, хотите привести широкий датафрейм к длинному формату, вы можете воспользоваться методом *`unpivot()`*. Этот метод принимает аргументы, показанные в табл. 15.2.

***Таблица 15.2. Аргументы метода*** *`unpivot()`*

| Аргумент | Описание |
|---|---|
| `Index` | Столбцы, данные из которых будут использоваться как идентификаторы для строк. Они останутся в результирующем датафрейме |
| `On` | Столбцы, содержащие имена колонок, которые будут свернуты вместе. Если аргумент не указан, будут использованы все столбцы, не вошедшие в аргумент `index` |
| `variable_name` | Имя будущего столбца, который будет хранить все имена свернутых столбцов |
| `value_name` | Имя будущего столбца, который будет хранить все значения свернутых столбцов |

Столбцы, отсутствующие в аргументах `index` и `on`, если они оба указаны, не будут включены в результирующий датафрейм.

Давайте проиллюстрируем это на примере. Возьмем за основу широкий датафрейм с отчетностью по студентам из предыдущего примера. Каждому студенту в этом датафрейме соответствует одна строка с оценками по предметам:

```
grades_wide = pl.DataFrame(
    {
        "student": ["Jeroen", "Thijs", "Ritchie"],
        "math": [85, 78, 92],
        "science": [90, 82, 85],
        "history": [88, 80, 87],
    }
)

grades_wide
```

Вывод:

```
shape: (3, 4)
┌─────────┬──────┬─────────┬─────────┐
│ student ┆ math ┆ science ┆ history │
│ ---     ┆ ---  ┆ ---     ┆ ---     │
│ str     ┆ i64  ┆ i64     ┆ i64     │
╞═════════╪══════╪═════════╪═════════╡
│ Jeroen  ┆ 85   ┆ 90      ┆ 88      │
│ Thijs   ┆ 78   ┆ 82      ┆ 80      │
│ Ritchie ┆ 92   ┆ 85      ┆ 87      │
└─────────┴──────┴─────────┴─────────┘
```

Вы можете развернуть этот датафрейм в длинный формат, в котором каждая строка будет соответствовать одной оценке студента по конкретному предмету:

```
grades_wide.unpivot(
    index=["student"],
    on=["math", "science", "history"],
    variable_name="subject",
    value_name="grade",
)
```

Вывод:

```
shape: (9, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
│ ---     ┆ ---     ┆ ---   │
│ str     ┆ str     ┆ i64   │
╞═════════╪═════════╪═══════╡
│ Jeroen  ┆ math    ┆ 85    │
│ Thijs   ┆ math    ┆ 78    │
│ Ritchie ┆ math    ┆ 92    │
```

```
| Jeroen  | science | 90    |
| Thijs   | science | 82    |
| Ritchie | science | 85    |
| Jeroen  | history | 88    |
| Thijs   | history | 80    |
| Ritchie | history | 87    |
└─────────┴─────────┴───────┘
```

При развороте мы идентифицировали строки по столбцу `student`, в котором содержатся имена студентов. В аргументе `on` мы перечислили имена столбцов, которые должны быть объединены в одной колонке с именем, переданным в виде аргумента `variable_name`. В аргументе `value_name` мы передали имя для столбца со значениями.

Рассмотрим еще один пример:

```
df = pl.DataFrame(
    {
        "student": ["Jeroen", "Thijs", "Ritchie", "Jeroen", "Thijs", "Ritchie"],
        "class": [
            "Math101",
            "Math101",
            "Math101",
            "Math102",
            "Math102",
            "Math102",
        ],
        "age": [20, 21, 22, 20, 21, 22],
        "semester": ["Fall", "Fall", "Fall", "Spring", "Spring", "Spring"],
        "math": [85, 78, 92, 88, 79, 95],
        "science": [90, 82, 85, 92, 81, 87],
        "history": [88, 80, 87, 85, 82, 89],
    }
)

df
```

Вывод:

```
shape: (6, 7)
┌─────────┬─────────┬─────┬──────────┬──────┬─────────┬─────────┐
| student | class   | age | semester | math | science | history |
| ---     | ---     | --- | ---      | ---  | ---     | ---     |
| str     | str     | i64 | str      | i64  | i64     | i64     |
╞═════════╪═════════╪═════╪══════════╪══════╪═════════╪═════════╡
| Jeroen  | Math101 | 20  | Fall     | 85   | 90      | 88      |
| Thijs   | Math101 | 21  | Fall     | 78   | 82      | 80      |
| Ritchie | Math101 | 22  | Fall     | 92   | 85      | 87      |
| Jeroen  | Math102 | 20  | Spring   | 88   | 92      | 85      |
| Thijs   | Math102 | 21  | Spring   | 79   | 81      | 82      |
| Ritchie | Math102 | 22  | Spring   | 95   | 87      | 89      |
└─────────┴─────────┴─────┴──────────┴──────┴─────────┴─────────┘
```

```
df.unpivot(
    index=["student", "class", "age", "semester"],
    on=["math", "science", "history"],
    variable_name="subject",
    value_name="grade",
)
```

Вывод:

```
shape: (18, 6)
┌─────────┬─────────┬─────┬──────────┬─────────┬───────┐
│ student ┆ class   ┆ age ┆ semester ┆ subject ┆ grade │
│ ---     ┆ ---     ┆ --- ┆ ---      ┆ ---     ┆ ---   │
│ str     ┆ str     ┆ i64 ┆ str      ┆ str     ┆ i64   │
╞═════════╪═════════╪═════╪══════════╪═════════╪═══════╡
│ Jeroen  ┆ Math101 ┆ 20  ┆ Fall     ┆ math    ┆ 85    │
│ Thijs   ┆ Math101 ┆ 21  ┆ Fall     ┆ math    ┆ 78    │
│ Ritchie ┆ Math101 ┆ 22  ┆ Fall     ┆ math    ┆ 92    │
│ Jeroen  ┆ Math102 ┆ 20  ┆ Spring   ┆ math    ┆ 88    │
│ Thijs   ┆ Math102 ┆ 21  ┆ Spring   ┆ math    ┆ 79    │
│ Ritchie ┆ Math102 ┆ 22  ┆ Spring   ┆ math    ┆ 95    │
│ Jeroen  ┆ Math101 ┆ 20  ┆ Fall     ┆ science ┆ 90    │
│ Thijs   ┆ Math101 ┆ 21  ┆ Fall     ┆ science ┆ 82    │
│ Ritchie ┆ Math101 ┆ 22  ┆ Fall     ┆ science ┆ 85    │
│ Jeroen  ┆ Math102 ┆ 20  ┆ Spring   ┆ science ┆ 92    │
│ Thijs   ┆ Math102 ┆ 21  ┆ Spring   ┆ science ┆ 81    │
│ Ritchie ┆ Math102 ┆ 22  ┆ Spring   ┆ science ┆ 87    │
│ Jeroen  ┆ Math101 ┆ 20  ┆ Fall     ┆ history ┆ 88    │
│ Thijs   ┆ Math101 ┆ 21  ┆ Fall     ┆ history ┆ 80    │
│ Ritchie ┆ Math101 ┆ 22  ┆ Fall     ┆ history ┆ 87    │
│ Jeroen  ┆ Math102 ┆ 20  ┆ Spring   ┆ history ┆ 85    │
│ Thijs   ┆ Math102 ┆ 21  ┆ Spring   ┆ history ┆ 82    │
│ Ritchie ┆ Math102 ┆ 22  ┆ Spring   ┆ history ┆ 89    │
└─────────┴─────────┴─────┴──────────┴─────────┴───────┘
```

# Транспонирование

Если вы хотите поменять местами столбцы и строки по диагонали без сохранения определенных столбцов в качестве идентификаторов, вы можете воспользоваться методом датафрейма *transpose()*. Этот метод работает только применительно к традиционным датафреймам и принимает аргументы, перечисленные в табл. 15.3.

Пришло время для примера. Возьмем датафрейм из предыдущего раздела:

```
grades_wide = pl.DataFrame(
    {
        "student": ["Jeroen", "Thijs", "Ritchie"],
        "math": [85, 78, 92],
        "science": [90, 82, 85],
```

```
        "history": [88, 80, 87],
    }
)

grades_wide
```

Вывод:

```
shape: (3, 4)
```

| student | math | science | history |
|---|---|---|---|
| str | i64 | i64 | i64 |
| Jeroen | 85 | 90 | 88 |
| Thijs | 78 | 82 | 80 |
| Ritchie | 92 | 85 | 87 |

***Таблица 15.3. Аргументы метода*** *transpose()*

| Аргумент | Описание |
|---|---|
| include_header | Указывает на то, следует ли создать первый столбец в итоговом датафрейме, содержащий имена столбцов исходного датафрейма |
| header_name | Если аргументу include_header передано значение True, с помощью аргумента header_name можно задать имя первого столбца. По умолчанию используется имя column |
| column_names | С помощью этого аргумента вы можете передать список или любой другой итерируемый объект с именами столбцов в создаваемом датафрейме |

Теперь перевернем наш датафрейм по главной диагонали:

```
report_columns = (f"report_{i + 1}" for i, _ in enumerate(grades_wide.columns)) ❶

grades_wide.transpose(
    include_header=True,
    header_name="original_headers",
    column_names=report_columns,
)
```

❶ Поскольку столбцы в датафрейме не могут иметь одинаковые имена, мы сгенерировали уникальные имена с помощью индекса.

Вывод:

```
shape: (4, 4)
```

| original_headers | report_1 | report_2 | report_3 |
|---|---|---|---|
| str | str | str | str |
| student | Jeroen | Thijs | Ritchie |
| math | 85 | 78 | 92 |

```
| science          | 90       | 82       | 85       |
| history          | 88       | 80       | 87       |
└──────────────────┴──────────┴──────────┴──────────┘
```

Как видите, все столбцы стали строками, а имена исходных столбцов сохранились в колонке с именем `original_headers`.

# Разворачивание в строки

Если в столбцах вашего датафрейма содержатся списки или массивы, это уже не назовешь широким форматом хранения, но он пока еще и не длинный. При желании раскрыть вложенные структуры и перенести значения из них в отдельные строки можно воспользоваться удобным методом датафрейма *explode()*. В качестве аргументов этот метод принимает только имена столбцов, которые необходимо развернуть в строки. Давайте в нашем предыдущем примере соберем оценки студентов по предметам в списки, как показано ниже:

```
grades_nested = pl.DataFrame(
    {
        "student": ["Jeroen", "Thijs", "Ritchie"],
        "math": [[85, 90, 88], [78, 82, 80], [92, 85, 87]],
    }
)

grades_nested
```

Вывод:

```
shape: (3, 2)
┌─────────┬─────────────┐
| student | math        |
| ---     | ---         |
| str     | list[i64]   |
╞═════════╪═════════════╡
| Jeroen  | [85, 90, 88] |
| Thijs   | [78, 82, 80] |
| Ritchie | [92, 85, 87] |
└─────────┴─────────────┘
```

Для преобразования этого датафрейма в длинный формат вызовем метод `explode()` и передадим ему имя столбца `math`:

```
grades_nested.explode("math")
```

Вывод:

```
shape: (9, 2)
┌─────────┬──────┐
| student | math |
| ---     | ---  |
```

```
| str     | i64  |
╞═════════╪══════╡
| Jeroen  | 85   |
| Jeroen  | 90   |
| Jeroen  | 88   |
| Thijs   | 78   |
| Thijs   | 82   |
| Thijs   | 80   |
| Ritchie | 92   |
| Ritchie | 85   |
| Ritchie | 87   |
└─────────┴──────┘
```

А вот пример с несколькими столбцами:

```
grades_nested = pl.DataFrame(
    {
        "student": ["Jeroen", "Thijs", "Ritchie"],
        "math": [[85, 90, 88], [78, 82, 80], [92, 85, 87]],
        "science": [[85, 90, 88], [78, 82], [92, 85, 87]],
        "history": [[85, 90, 88], [78, 82], [92, 85, 87]],
    }
)

grades_nested
```

Вывод:

```
shape: (3, 4)
┌─────────┬─────────────┬─────────────┬─────────────┐
| student | math        | science     | history     |
| ---     | ---         | ---         | ---         |
| str     | list[i64]   | list[i64]   | list[i64]   |
╞═════════╪═════════════╪═════════════╪═════════════╡
| Jeroen  | [85, 90, 88] | [85, 90, 88] | [85, 90, 88] |
| Thijs   | [78, 82, 80] | [78, 82]     | [78, 82]     |
| Ritchie | [92, 85, 87] | [92, 85, 87] | [92, 85, 87] |
└─────────┴─────────────┴─────────────┴─────────────┘
```

И снова для преобразования этого датафрейма в длинный формат мы воспользуемся методом `explode()`. Но поскольку Тейс сдал не все тесты (классический Тейс), мы не сможем сделать этого. Дело в том, что разворачиваемые в строки столбцы должны содержать одинаковое количество элементов, иначе возникнет ошибка, показанная ниже:

```
grades_nested.explode("math", "science", "history")
```

Вывод:

```
ShapeError: exploded columns must have matching element counts
```

Сначала нам придется привести наш датафрейм к длинному формату с помощью метода `unpivot()`:

```
grades_nested_long = grades_nested.unpivot(
    index="student", variable_name="subject", value_name="grade"
)

grades_nested_long
```

Вывод:

```
shape: (9, 3)
┌─────────┬─────────┬──────────────┐
│ student ┆ subject ┆ grade        │
│ ---     ┆ ---     ┆ ---          │
│ str     ┆ str     ┆ list[i64]    │
╞═════════╪═════════╪══════════════╡
│ Jeroen  ┆ math    ┆ [85, 90, 88] │
│ Thijs   ┆ math    ┆ [78, 82, 80] │
│ Ritchie ┆ math    ┆ [92, 85, 87] │
│ Jeroen  ┆ science ┆ [85, 90, 88] │
│ Thijs   ┆ science ┆ [78, 82]     │
│ Ritchie ┆ science ┆ [92, 85, 87] │
│ Jeroen  ┆ history ┆ [85, 90, 88] │
│ Thijs   ┆ history ┆ [78, 82]     │
│ Ritchie ┆ history ┆ [92, 85, 87] │
└─────────┴─────────┴──────────────┘
```

Теперь мы можем воспользоваться методом `explode()`, как показано ниже:

```
grades_nested_long.explode("grade")
```

Вывод:

```
shape: (25, 3)
┌─────────┬─────────┬───────┐
│ student ┆ subject ┆ grade │
│ ---     ┆ ---     ┆ ---   │
│ str     ┆ str     ┆ i64   │
╞═════════╪═════════╪═══════╡
│ Jeroen  ┆ math    ┆ 85    │
│ Jeroen  ┆ math    ┆ 90    │
│ Jeroen  ┆ math    ┆ 88    │
│ Thijs   ┆ math    ┆ 78    │
│ Thijs   ┆ math    ┆ 82    │
│ …       ┆ …       ┆ …     │
│ Thijs   ┆ history ┆ 78    │
│ Thijs   ┆ history ┆ 82    │
│ Ritchie ┆ history ┆ 92    │
│ Ritchie ┆ history ┆ 85    │
│ Ritchie ┆ history ┆ 87    │
└─────────┴─────────┴───────┘
```

Обратите внимание, что при разворачивании в строки сразу нескольких столбцов порядок значений в списках имеет значение. Элементы, находя-

щиеся на одинаковых позициях в списках, будут собраны в одну строку в результирующем датафрейме. О сортировке списков мы говорили в главе 12.

Метод `explode()` также можно применять с вложенными списками, как показано ниже:

```
nested_lists = pl.DataFrame(
    {
        "id": [1, 2],
        "nested_value": [[["a", "b"]], [["c"], ["d", "e"]]],
    },
    strict=False,
)

nested_lists
```

Вывод:

```
shape: (2, 2)
┌─────┬──────────────────────┐
│ id  ┆ nested_value         │
│ --- ┆ ---                  │
│ i64 ┆ list[list[str]]      │
╞═════╪══════════════════════╡
│ 1   ┆ [["a", "b"]]         │
│ 2   ┆ [["c"], ["d", "e"]]  │
└─────┴──────────────────────┘
```

Обратите внимание, что в этом случае разворачивание в строки будет производиться по одному уровню вложенности за раз[1]:

```
nested_lists.explode("nested_value")
```

Вывод:

```
shape: (3, 2)
┌─────┬──────────────┐
│ id  ┆ nested_value │
│ --- ┆ ---          │
│ i64 ┆ list[str]    │
╞═════╪══════════════╡
│ 1   ┆ ["a", "b"]   │
│ 2   ┆ ["c"]        │
│ 2   ┆ ["d", "e"]   │
└─────┴──────────────┘
```

Если вы хотите добраться до самих строковых значений, вам необходимо применить метод `explode()` дважды:

```
nested_lists.explode("nested_value").explode("nested_value")
```

---

[1] Уровни вложенности в списках подобны слоям репчатого лука.

Вывод:

```
shape: (5, 2)
┌─────┬──────────────┐
│ id  ┆ nested_value │
│ --- ┆ ---          │
│ i64 ┆ str          │
╞═════╪══════════════╡
│ 1   ┆ a            │
│ 1   ┆ b            │
│ 2   ┆ c            │
│ 2   ┆ d            │
│ 2   ┆ e            │
└─────┴──────────────┘
```

# Партиционирование датафреймов

В главе 13 мы подробно обсуждали метод `group_by()`. Похожий метод есть для разделения исходного датафрейма на отдельные *партиции* (partition), представляющие собой также датафреймы. Используя метод *`partition_by()`*, вы группируете датафрейм по указанным столбцам, но результирующие группы возвращаете в виде отдельных датафреймов. В табл. 15.4 представлен список аргументов метода `partition_by()`.

***Таблица 15.4. Аргументы метода*** *`partition_by()`*

| Аргумент | Описание |
|---|---|
| `by` и `*more_by` | Столбцы для группировки |
| `maintain_order` | Обеспечивает детерминированность порядка групп |
| `include_key` | Вместо списка датафреймов возвращает список кортежей с ключами групп и датафреймами |
| `as_dict` | Возвращает результат в виде словаря |

Давайте построим небольшой датафрейм с продажами по регионам:

```
sales = pl.DataFrame(
    {
        "OrderID": [1, 2, 3, 4, 5, 6],
        "Product": ["A", "B", "A", "C", "B", "A"],
        "Quantity": [10, 5, 8, 7, 3, 12],
        "Region": ["North", "South", "North", "West", "South", "West"],
    }
)
```

Теперь можно партиционировать этот датафрейм по столбцу `Region`, как показано ниже:

```
sales.partition_by("Region")
```

Вывод:

```
[shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 1       ┆ A       ┆ 10       ┆ North  │
│ 3       ┆ A       ┆ 8        ┆ North  │
└─────────┴─────────┴──────────┴────────┘,
shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 2       ┆ B       ┆ 5        ┆ South  │
│ 5       ┆ B       ┆ 3        ┆ South  │
└─────────┴─────────┴──────────┴────────┘,
shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 4       ┆ C       ┆ 7        ┆ West   │
│ 6       ┆ A       ┆ 12       ┆ West   │
└─────────┴─────────┴──────────┴────────┘]
```

Если вам нужно удалить столбец, по которому выполнялось партиционирование, вы можете передать аргументу `include_key` значение `False`:

```
sales.partition_by("Region", include_key=False)
```

Вывод:

```
[shape: (2, 3)
┌─────────┬─────────┬──────────┐
│ OrderID ┆ Product ┆ Quantity │
│ ---     ┆ ---     ┆ ---      │
│ i64     ┆ str     ┆ i64      │
╞═════════╪═════════╪══════════╡
│ 1       ┆ A       ┆ 10       │
│ 3       ┆ A       ┆ 8        │
└─────────┴─────────┴──────────┘,
shape: (2, 3)
┌─────────┬─────────┬──────────┐
│ OrderID ┆ Product ┆ Quantity │
│ ---     ┆ ---     ┆ ---      │
│ i64     ┆ str     ┆ i64      │
╞═════════╪═════════╪══════════╡
```

```
| 2        | B        | 5        |
| 5        | B        | 3        |
└──────────┴──────────┴──────────┘,
shape: (2, 3)
┌──────────┬──────────┬──────────┐
| OrderID  | Product  | Quantity |
| ---      | ---      | ---      |
| i64      | str      | i64      |
╞══════════╪══════════╪══════════╡
| 4        | C        | 7        |
| 6        | A        | 12       |
└──────────┴──────────┴──────────┘]
```

Наконец, если вы хотите получить результат в виде словаря с кортежем ключей групп в качестве ключей и датафреймами в качестве значений, вы можете передать аргументу `as_dict` метода `partition_by()` значение `True`:

```
sales_dict = sales.partition_by(["Region"], as_dict=True)

sales_dict
```

Вывод:

```
{('North',): shape: (2, 4)
┌──────────┬──────────┬──────────┬──────────┐
| OrderID  | Product  | Quantity | Region   |
| ---      | ---      | ---      | ---      |
| i64      | str      | i64      | str      |
╞══════════╪══════════╪══════════╪══════════╡
| 1        | A        | 10       | North    |
| 3        | A        | 8        | North    |
└──────────┴──────────┴──────────┴──────────┘,
('South',): shape: (2, 4)
┌──────────┬──────────┬──────────┬──────────┐
| OrderID  | Product  | Quantity | Region   |
| ---      | ---      | ---      | ---      |
| i64      | str      | i64      | str      |
╞══════════╪══════════╪══════════╪══════════╡
| 2        | B        | 5        | South    |
| 5        | B        | 3        | South    |
└──────────┴──────────┴──────────┴──────────┘,
('West',): shape: (2, 4)
┌──────────┬──────────┬──────────┬──────────┐
| OrderID  | Product  | Quantity | Region   |
| ---      | ---      | ---      | ---      |
| i64      | str      | i64      | str      |
╞══════════╪══════════╪══════════╪══════════╡
| 4        | C        | 7        | West     |
| 6        | A        | 12       | West     |
└──────────┴──────────┴──────────┴──────────┘}
```

После этого вы можете обращаться к отдельным датафреймам по ключу, как показано ниже:

```
sales_dict[("North",)]
```

Вывод:

```
shape: (2, 4)
┌─────────┬─────────┬──────────┬────────┐
│ OrderID ┆ Product ┆ Quantity ┆ Region │
│ ---     ┆ ---     ┆ ---      ┆ ---    │
│ i64     ┆ str     ┆ i64      ┆ str    │
╞═════════╪═════════╪══════════╪════════╡
│ 1       ┆ A       ┆ 10       ┆ North  │
│ 3       ┆ A       ┆ 8        ┆ North  │
└─────────┴─────────┴──────────┴────────┘
```

# Заключение

В этой главе мы поговорили о возможностях изменения формы ваших данных.

Вы узнали, что:

- существуют длинный и широкий форматы табличных данных;
- для приведения данных из длинного формата в широкий можно воспользоваться методом датафрейма `pivot()`;
- для приведения данных обратно из широкого формата в длинный существует метод датафрейма `unpivot()`;
- для транспонирования датафрейма по диагонали можно вызвать метод `transpose()`;
- для распаковки вложенных значений в длинный формат по слоям можно вызвать метод `explode()`;
- для разделения датафрейма на несколько датафреймов по группам служит метод `partition_by()`.

Теперь, когда вы умеете приводить табличные данные к различным форматам, вы можете пробовать их визуализировать. Об этом мы и поговорим в следующей главе.

# ЧАСТЬ V

# Дополнительные возможности

# Глава 16

# Визуализация данных

В предыдущих главах мы рассмотрели все необходимые инструменты, имеющиеся в Polars для преобразования сырых необработанных данных в хорошо подготовленные и продуманные датафреймы. Но как на основе этих датафреймов сделать правильные выводы?

Один из способов состоит в визуализации данных, и в Python для этой цели существует великое множество специальных библиотек. К примеру, библиотека Matplotlib часто используется для создания несложных графиков, пакет hvPlot может пригодиться для быстрой визуализации, Bokeh – для построения интерактивных визуализаций, plotnine – для использования принципов так называемой *грамматики графики* (Grammar of Graphics) в Python, а Altair – для применения встроенных графических возможностей в Polars. На рис. 16.1 представлена вся вселенная визуализации в Python.

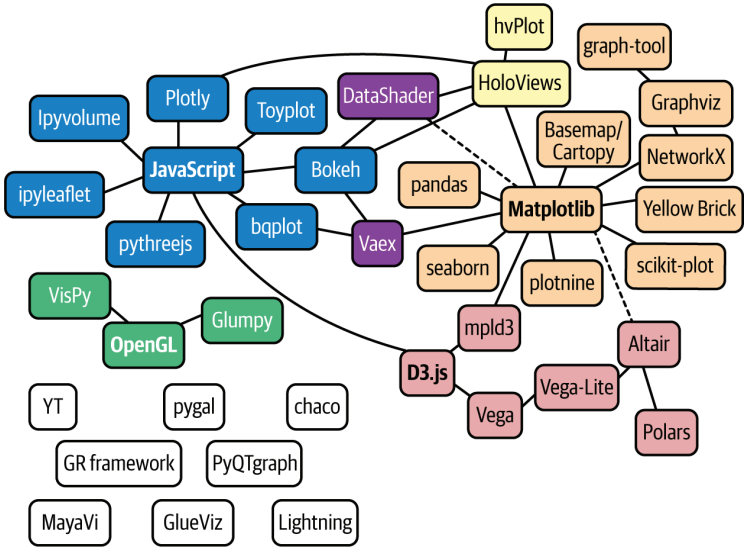



**Рис. 16.1** ❖ Графические пакеты для Python и зависимости между ними (переработанная версия диаграммы Джейка Вандерпласа (Jake VanderPlas) с согласия автора)

На эту диаграмму даже смотреть больно. И велика вероятность, что при необходимости воспользоваться одной из представленных библиотек вы даже не сможете найти из них наиболее подходящую. Более того, все эти библиотеки обладают своими ограничениями и недостатками.

Визуализация данных не ограничивается одним лишь выводом на экран веселых картинок. Это вполне себе самостоятельная область науки о данных. Преобразовывая датафреймы в графический вид, вы обретаете возможность отслеживать тенденции в данных, идентифицировать выбросы и выстраивать целые истории, помогающие в принятии решений. Эффективная визуализация способна помочь сделать непонятное понятным, мутное – ясным, а сложное – простым. В результате вы сможете принимать более взвешенные решения на основе данных.

**Это далеко не все о визуализации**

Как вы понимаете, визуализация данных представляет собой отдельную большую область знаний, которую невозможно охватить в одной главе книги. Мы осветим лишь малую ее часть, касающуюся представления данных в графическом виде в Polars. При этом мы не будем пытаться охватить все библиотеки, а сосредоточимся лишь на нескольких, чтобы повествование было более ясным и понятным. Также мы не будем проводить сравнительный анализ разных графических пакетов.

Одной из важнейших составляющих области визуализации данных является выбор диаграмм, наилучшим образом отражающих тот или иной разрез анализа. Более детальную информацию о видах элементов визуализации можно почерпнуть, например, из книги *Fundamentals of Data Visualization* Клауса Вильке (Claus Wilke), особенно из главы 5, посвященной разным видам графиков и диаграмм.

В этой главе вы узнаете, как:

- быстро строить столбчатые и точечные диаграммы, графики плотности распределения и гистограммы с помощью встроенного функционала Polars на основе Altair;
- компоновать и разделять по слоям несколько диаграмм;
- создавать интерактивные визуализации;
- выводить миллионы точек на карте;
- использовать альтернативные графические библиотеки, такие как hvPlot и plotnine;
- создавать красивые таблицы со встроенными графиками с помощью пакета Great Tables.

Прочитав эту главу, вы будете хорошо понимать предназначение библиотек Altair, hvPlot, plotnine и Great Tables и научитесь использовать их совместно с Polars. Но сначала рассмотрим набор данных, с которым будем работать.

Все инструкции по загрузке и установке требуемых файлов вы найдете в главе 2. Мы будем исходить из предположения о том, что все рабочие файлы вы поместили в директорию *data*.

# Поездки на велосипедах в Нью-Йорке

На протяжении этой главы мы будем использовать уже знакомый вам по первой главе набор данных, относящийся к поездкам на прокатных велосипедах в Нью-Йорке. Давайте освежим информацию в памяти[1]:

```
trips = pl.read_parquet("data/citibike/*.parquet")
print(trips[:, :4])
print(trips[:, 4:7])
print(trips[:, 7:11])
print(trips[:, 11:])
```

Вывод:

```
shape: (2_638_971, 4)
```

| bike_type | rider_type | datetime_start | datetime_end |
|---|---|---|---|
| cat | cat | datetime[μs] | datetime[μs] |
| electric | member | 2024-03-01 00:00:02.490 | 2024-03-01 00:27:39.295 |
| electric | member | 2024-03-01 00:00:04.120 | 2024-03-01 00:09:29.384 |
| … | … | … | … |
| electric | member | 2024-03-31 23:55:41.173 | 2024-03-31 23:57:25.079 |
| electric | member | 2024-03-31 23:57:16.025 | 2024-03-31 23:59:22.134 |

```
shape: (2_638_971, 3)
```

| duration | station_start | station_end |
|---|---|---|
| duration[μs] | str | str |
| 27m 36s 805ms | W 30 St & 8 Ave | Maiden Ln & Pearl St |
| 9m 25s 264ms | Longwood Ave & Southern Blvd | Lincoln Ave & E 138 St |
| … | … | … |
| 1m 43s 906ms | S 4 St & Wythe Ave | S 3 St & Bedford Ave |
| 2m 6s 109ms | Montrose Ave & Bushwick Ave | Humboldt St & Varet St |

```
shape: (2_638_971, 4)
```

| neighborhood_start | neighborhood_end | borough_start | borough_end |
|---|---|---|---|
| str | str | str | str |

[1] Мы выводим датафрейм по частям, чтобы он поместился на страницах книги. С полным датафреймом и всеми исходными кодами можно ознакомиться в нашем публичном репозитории по адресу *https://github.com/jeroenjanssens/python-polars-the-definitive-guide*.

```
| Chelsea           | Financial District | Manhattan     | Manhattan   |
| Longwood          | Mott Haven         | Bronx         | Bronx       |
| …                 | …                  | …             | …           |
| Williamsburg      | Williamsburg       | Brooklyn      | Brooklyn    |
| Williamsburg      | Williamsburg       | Brooklyn      | Brooklyn    |
└───────────────────┴────────────────────┴───────────────┴─────────────┘
shape: (2_638_971, 5)
┌────────────┬─────────────┬────────────┬─────────────┬──────────┐
| lat_start  | lon_start   | lat_end    | lon_end     | distance |
| ---        | ---         | ---        | ---         | ---      |
| f64        | f64         | f64        | f64         | f64      |
╞════════════╪═════════════╪════════════╪═════════════╪══════════╡
| 40.749614  | -73.995071  | 40.707065  | -74.007319  | 4.842569 |
| 40.816459  | -73.896576  | 40.810893  | -73.927311  | 2.659582 |
| …          | …           | …          | …           | …        |
| 40.712996  | -73.965971  | 40.712605  | -73.962644  | 0.283781 |
| 40.707678  | -73.940297  | 40.703172  | -73.940636  | 0.501835 |
└────────────┴─────────────┴────────────┴─────────────┴──────────┘
```

В марте 2024 года на велосипедах Citi Bike было совершено более 2.6 млн поездок по четырем округам Нью-Йорка: Бронкс (The Bronx), Бруклин (Brooklyn), Манхэттен (Manhattan) и Куинс (Queens). Как мы помним, в округе Стейтен Айленд (Staten Island) отсутствуют станции проката Citi Bike. При этом каждый округ содержит множество районов.

В нашем датафрейме в столбце `bike_type` могут присутствовать значения `electric` и `classic`, а в столбце `rider_type` – `member` или `casual`. В столбце `duration` хранится разница между значениями в полях `datetime_start` и `datetime_end`.

В столбцах `lat_start`, `lon_start`, `lat_end` и `lon_end` содержатся начальная и конечная координаты поездок соответственно. Расстояние в столбце `distance` выражено в километрах кратчайшего пути – по прямой.

В нашем датафрейме `trips` содержится большое количество столбцов, включая дату и время, категории, названия и координаты. Это позволит нам воспользоваться разными видами визуализации.

Что ж, давайте вместе узнаем, когда жители и гости Нью-Йорка катаются на велосипедах, куда они ездят и какие станции пользуются наибольшей популярностью. И все это с помощью визуализации.

# Встроенные графические возможности на основе Altair

Самым простым и быстрым способом вывести данные из датафреймов Polars в графическом виде является использование встроенных методов библиотеки. Эти методы доступны в пространствах имен *`df.plot`* и *`series.plot`*. Примерами таких методов являются `df.plot.scatter()` и `series.plot.kde()`. Под капотом эти методы перенаправляются на исполнение библиотеке Altair.

**Графическая нестабильность**

На момент написания книги встроенные графические методы в библиотеке Polars помечены как нестабильные, а это означает, что они могут претерпевать изменения. На самом деле на заре Polars в качестве основного бэкенда графического движка использовалась библиотека hvPlot, и лишь через несколько месяцев выбор был сделан в пользу Altair. Если выбор снова изменится, вы всегда сможете использовать для своих графиков Altair напрямую. Помните, что встроенные графические функции представлены только для удобства и не добавляют никакого функционала лежащим в их основе методам из библиотеки Altair.

# Знакомство с Altair

*Altair* (*https://altair-viz.github.io*) представляет собой пакет декларативной статистической визуализации в Python, предназначенный для создания интерактивных визуализаций с минимумом написанного кода. Он был разработан Джейком Вандерпласом на основе языка высокого уровня для описания интерактивной графики Vega-Lite. Altair предоставляет мощный, но простой в использовании интерфейс для создания богатого спектра визуализаций, позволяющий сконцентрироваться на данных, а не на сложностях процесса визуализации.

Если при инсталляции Polars вы установили дополнительные зависимости, воспользовавшись ключевыми словами `all` или `plot`, значит, у вас уже есть Altair (см. главу 2, в которой мы обсуждали все тонкости установки Polars). Также вы можете установить Altair отдельно с помощью следующей команды:

```
$ uv pip install altair
```

При использовании встроенных методов вам обычно не требуется явно импортировать пакет Altair. Но сделать это при необходимости можно так:

```
import altair as alt
```

Философия библиотеки Altair основана на принципах декларативной визуализации. Это значит, что вы должны указать, что именно хотите визуализировать, а не как именно это нужно сделать.

Ключевые концепции Altair:

- *визуализация на основе данных*: Altair фокусируется на данных, позволяя вам напрямую указывать преобразования и кодировку данных;
- *декларативный синтаксис*: вы задаете визуальные свойства, такие как цвет, форму и размер, и их связь с данными, а о выводе не беспокоитесь;
- *интерактивность*: библиотека Altair поддерживает интерактивные особенности, такие как выноски, фильтрация и динамические обновления, что повышает потенциал визуализаций;
- *интеграция с Polars*: Altair бесшовно работает с датафреймами Polars без промежуточного звена в виде pandas благодаря пакету Narwhals.

## Методы пространств имен plot

Чтобы узнать, какие методы доступны в двух пространствах имен `plot`, вы можете воспользоваться клавишей табуляции, например нажав на нее после написания `df.plot.`:

```
trips.plot.<TAB>
```

Доступные методы в пространстве имен `df.plot`:

- *`df.plot.bar()`*: выводит столбчатую диаграмму с накоплениями или группами;
- *`df.plot.line()`*: выводит линейный график, как для временных рядов;
- *`df.plot.point()`*: выводит точечную диаграмму, или диаграмму рассеяния, для сравнения двух переменных;
- *`df.plot.scatter()`*: синоним метода `df.plot.point()`.

Автодополнение при помощи клавиши табуляции будет корректно работать, только если вы заранее присвоите объект Series переменной, как показано ниже:

```
bike_type = trips.get_column("bike_type")
bike_type.plot<TAB>
```

Доступные методы в пространстве имен `series.plot`:

- *`series.plot.hist()`*: выводит распределение на одной или нескольких гистограммах с набором столбиков;
- *`series.plot.density()`*: выводит график плотности для одной или нескольких переменных;
- *`series.plot.line()`*: выводит линейный график, как для временных рядов.

Давайте опробуем на практике некоторые из этих методов.

## Графический анализ датафреймов

Начнем с точечной диаграммы, с помощью которой посмотрим, как зависят друг от друга дистанция и длительность поездок на классических и электрических велосипедах:

```
trips_speed = trips.select(
    pl.col("distance"),
    pl.col("duration").dt.total_seconds() / 3600, ❶
    pl.col("bike_type"),
).with_columns(speed=pl.col("distance") / pl.col("duration"))

trips_speed
```

❶ Длительность поездок в столбце `duration` выражена в часах. Мы могли бы воспользоваться методом `Expr.dt.total_hours()`, но он возвращает количество полных часов.

Вывод:

```
shape: (2_638_971, 4)
┌──────────┬──────────┬──────────┬──────────┐
│ distance ┆ duration ┆ bike_type ┆ speed    │
│ ---      ┆ ---      ┆ ---       ┆ ---      │
│ f64      ┆ f64      ┆ cat       ┆ f64      │
╞══════════╪══════════╪══════════╪══════════╡
│ 4.842569 ┆ 0.46     ┆ electric  ┆ 10.527324 │
│ 2.659582 ┆ 0.156944 ┆ electric  ┆ 16.94601  │
│ …        ┆ …        ┆ …         ┆ …         │
│ 0.283781 ┆ 0.028611 ┆ electric  ┆ 9.918543  │
│ 0.501835 ┆ 0.035    ┆ electric  ┆ 14.338131 │
└──────────┴──────────┴──────────┴──────────┘
```

Вы можете создать точечный график в Altair, воспользовавшись методом `df.plot.scatter()`, как показано ниже:

```
trips_speed.plot.scatter(
    x="distance",
    y="duration",
    color="bike_type:N",
)
```

Вывод:

```
MaxRowsError: The number of rows in your dataset is greater than the maximum all owed
(5000).

Try enabling the VegaFusion data transformer which raises this limit by pre-evaluating data
transformations in Python.
    >> import altair as alt
    >> alt.data_transformers.enable("vegafusion")

Or, see https://altair-viz.github.io/user_guide/large_datasets.html for addition
al information
on how to plot large datasets.
alt.Chart(...)
```

Упс, и правда, Altair терпеть не может большие датафреймы. В следующем разделе мы посмотрим, как можно обойти это ограничение. Сейчас же давайте ограничимся поездками с определенной станции:

```
trips_speed = (
    trips.filter(pl.col("station_start") == "W 70 St & Amsterdam Ave")
    .select(
        pl.col("distance"),
        pl.col("duration").dt.total_seconds() / 3600,
        pl.col("bike_type"),
    )
    .with_columns(speed=pl.col("distance") / pl.col("duration"))
)

trips_speed
```

Вывод:

```
shape: (4_963, 4)
```

| distance | duration | bike_type | speed |
|---|---|---|---|
| f64 | f64 | cat | f64 |
| 0.596344 | 0.039167 | electric | 15.225797 |
| 1.134568 | 0.153611 | classic | 7.385977 |
| … | … | … | … |
| 6.784337 | 0.358611 | electric | 18.918368 |
| 2.077397 | 0.1375 | electric | 15.108345 |

Теперь у нас осталось меньше 5000 записей. Давайте снова попробуем вывести точечный график:

```
trips_speed.plot.scatter(
    x="distance",
    y="duration",
    color="bike_type:N", ❶
)
```

❶ Суффикс `:N` после имени столбца `bike_type` позволяет Altair интерпретировать его как номинальное значение, что гарантирует подходящую раскраску двух групп, как показано на рис. 16.2.

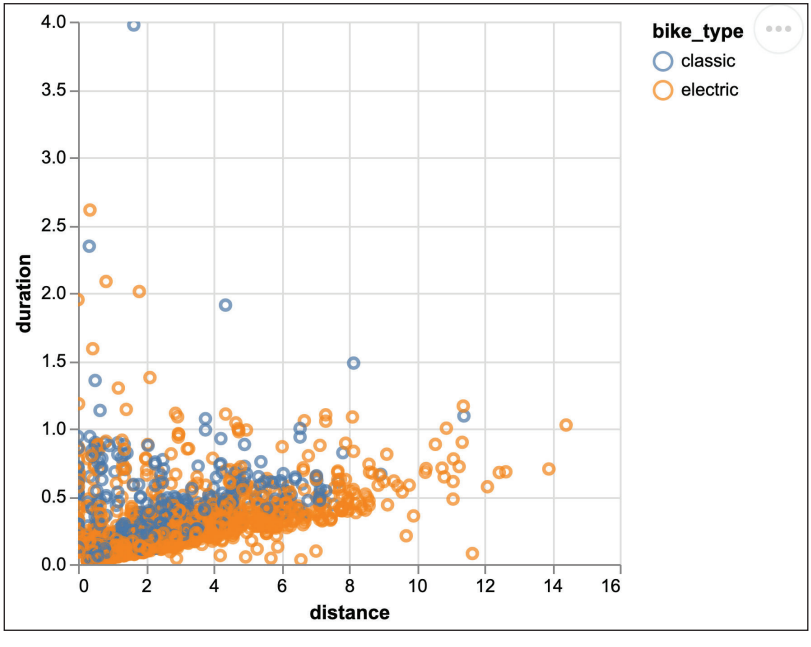



**Рис. 16.2** ❖ Точечная диаграмма, построенная при помощи библиотеки Altair

## Ограничение на размер

В библиотеке Altair по умолчанию стоит ограничение на 5000 строк в датафрейме, что предотвращает проблемы при работе с объемными данными в браузере. На самом деле это ограничение продиктовано даже не самой библиотекой Altair, а языком Vega-Lite. Но вы можете обойти его, если необходимо. Для этого нужно изменить параметр `max_rows` следующим образом:

```
import altair as alt

alt.data_transformers.disable_max_rows()
```

В результате ограничение будет снято, что позволит Altair работать с большими датафреймами. Но будьте осторожны, поскольку обработка объемных данных может негативно сказаться на производительности системы, особенно при работе в браузере.

Лучше для решения этой проблемы было бы воспользоваться VegaFusion. *VegaFusion* представляет собой внешний проект, эффективно использующий движок на языке Rust для преобразования данных, например для разбиения на группы или агрегации. Это позволяет снизить размер датафрейма и избавиться от лишних столбцов. При включенной опции VegaFusion этот движок предвычисляет все преобразования во время отображения графиков, сохранения или конвертации в словари или JSON.

Установить VegaFusion можно следующим образом:

```
$ uv pip install "vegafusion[embed]"
```

Для активации этой опции нужно выполнить показанную ниже инструкцию:

```
alt.data_transformers.enable("vegafusion")
```

После активации VegaFusion вы сможете работать с датафреймами, достигающими 100 тысяч строк. Ограничение на количество строк применяется после выполнения всех преобразований данных, так что для визуализаций, подобных гистограмме, это ограничение вряд ли станет проблемой.

Также вы можете агрегировать свои исходные данные самостоятельно. К примеру, если вы хотите создать столбчатую диаграмму с накоплением, вам понадобится всего несколько строк, а не все их исходное количество:

```
trips_type_counts = trips.group_by("rider_type", "bike_type").len()
trips_type_counts
```

Вывод:

```
shape: (4, 3)
┌────────────┬────────────┬─────────┐
| rider_type | bike_type | len     |
| ---        | ---       | ---     |
| cat        | cat       | u32     |
```

```
┌─────────────┬─────────────┬──────────┐
│ member      │ electric    │ 1412598  │
│ casual      │ electric    │ 294679   │
│ member      │ classic     │ 811168   │
│ casual      │ classic     │ 120526   │
└─────────────┴─────────────┴──────────┘
```

На таком агрегированном наборе данных Altair без проблем построит диаграмму, показанную на рис. 16.3.

```
trips_type_counts.plot.bar(
    x="rider_type", y="len", fill="bike_type:N"
).properties(
    width=300,
)
```

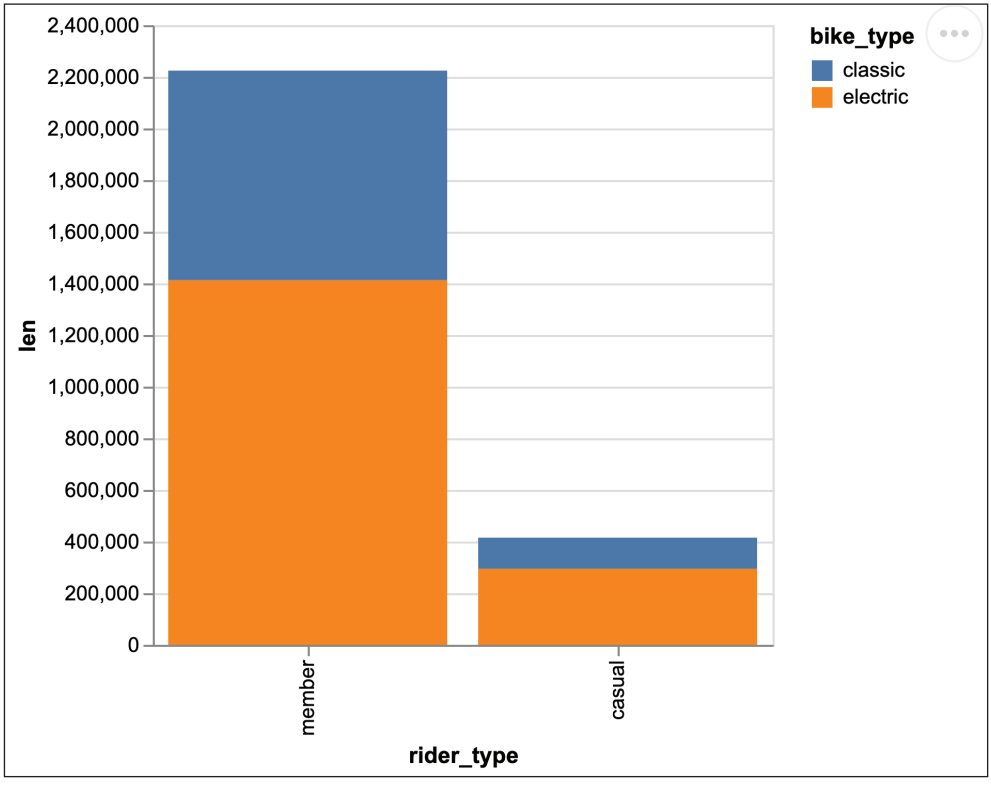



**Рис. 16.3** ❖ Столбчатая диаграмма с накоплением, построенная при помощи Altair

Если ни одна из предложенных выше стратегий вам не подходит, вы можете воспользоваться для визуализации данных другими пакетами, такими как hvPlot или plotnine.

# Визуализация объектов Series

Как мы уже упоминали выше, некоторые типы визуализации доступны только в пространстве имен `series.plot`. Ниже приведен пример создания графика плотности распределения на основе столбца `distance`. Результат показан на рис. 16.4.

```
trips_speed["distance"].plot.kde()
```

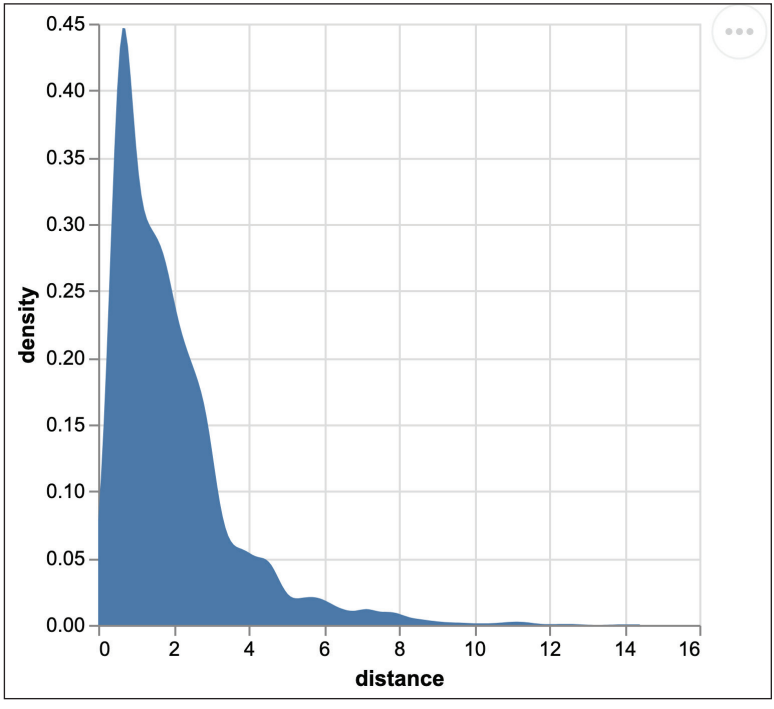



**Рис. 16.4.** График плотности распределения, построенный при помощи Altair

Столь же просто вы можете построить и гистограмму:

```
trips_speed["distance"].plot.hist()
```

Вывод показан на рис. 16.5.

Давайте также построим обычный линейный график. Для этого сначала сгруппируем данные по часам и в качестве агрегаций рассчитаем количество поездок и среднюю скорость:

```
trips_hour_num_speed = (
    trips.sort("datetime_start")
    .group_by_dynamic("datetime_start", every="1h")
    .agg(
```

```
        num_trips=pl.len(),
        speed=(
            pl.col("distance") / (pl.col("duration").dt.total_seconds() / 3600)
        ).median(),
    )
    .filter(pl.col("datetime_start") > pl.date(2024, 3, 26))
)

trips_hour_num_speed
```

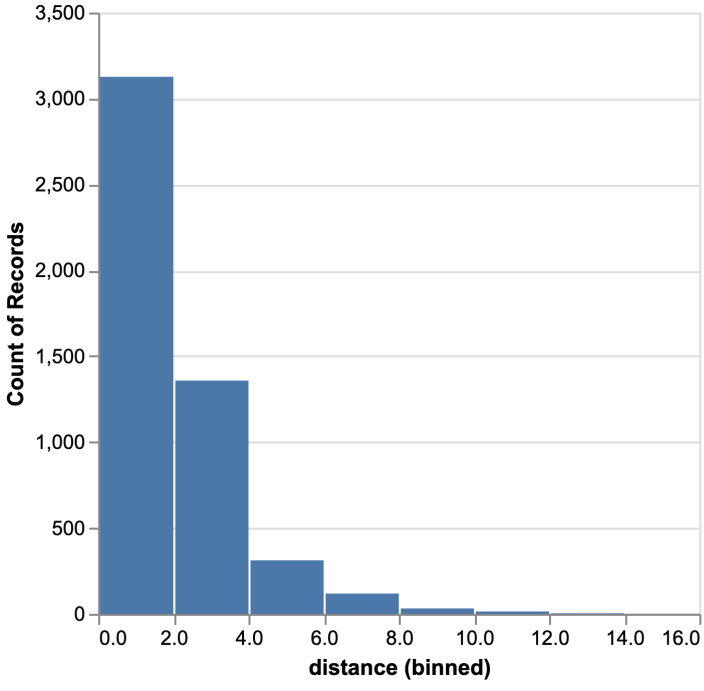



**Рис. 16.5** ❖ Гистограмма, построенная при помощи Altair

Вывод:

```
shape: (143, 3)
```

| datetime_start | num_trips | speed |
|---|---|---|
| datetime[μs] | u32 | f64 |
| 2024-03-26 01:00:00 | 295 | 13.813106 |
| 2024-03-26 02:00:00 | 181 | 14.317642 |
| 2024-03-26 03:00:00 | 119 | 13.150871 |
| 2024-03-26 04:00:00 | 222 | 14.239003 |
| 2024-03-26 05:00:00 | 878 | 13.966456 |
| … | … | … |

```
| 2024-03-31 19:00:00 | 5170      | 10.692934 |
| 2024-03-31 20:00:00 | 3640      | 11.057322 |
| 2024-03-31 21:00:00 | 2813      | 11.429875 |
| 2024-03-31 22:00:00 | 2333      | 11.432521 |
| 2024-03-31 23:00:00 | 1333      | 12.046345 |
└─────────────────────┴───────────┴───────────┘
```

Теперь можно легко построить линейный график при помощи метода `df.plot.line()`. Результат показан на рис. 16.6.

```
trips_hour_num_speed.plot.line(x="datetime_start", y="num_trips")
```

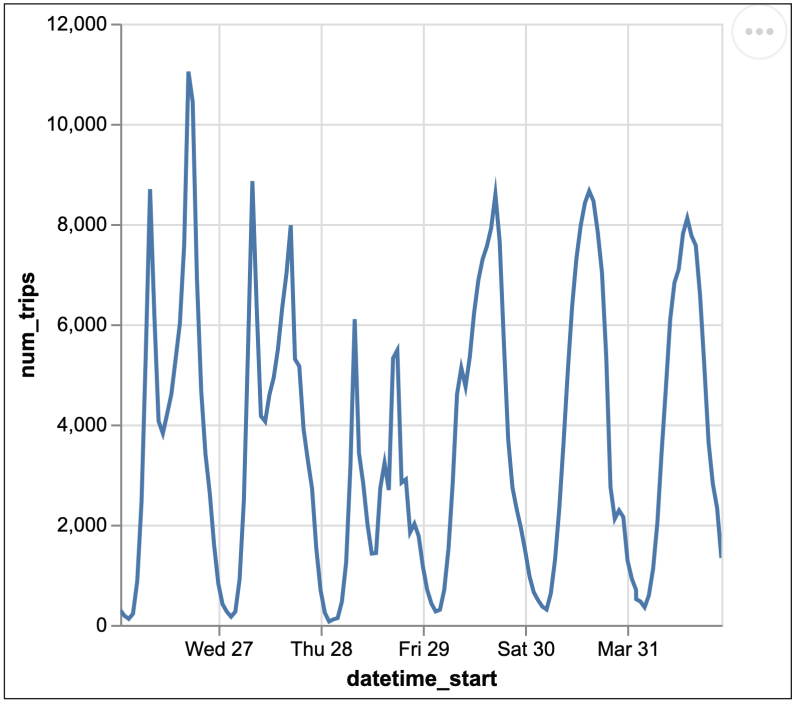



**Рис. 16.6** ❖ Линейный график, построенный при помощи Altair

Эти примеры демонстрируют, как можно использовать встроенные в Polars методы визуализации, обращающиеся под капотом к библиотеке Altair. Данные методы могут быть полезны для быстрого построения графиков, так называемых визуализаций ad hoc. В то же время у Altair есть и некоторые дополнительные возможности, такие как:

- создание визуализаций с несколькими слоями;
- объединение и связывание диаграмм;
- добавление выносок на графики и использование других средств интерактивности.

Но эти особенности Altair выходят за рамки данной книги. За дополнительной информацией вы можете обратиться к документации Altair по адресу *https://altair-viz.github.io*.

# Визуализация «как в Pandas» с помощью hvPlot

*hvPlot* отличается от большинства графических пакетов тем, что сам по себе он не создает никаких визуализаций. Вместо этого он предлагает универсальный интерфейс для трех других библиотек, а именно Bokeh, Matplotlib и Plotly. При этом API этого пакета написан в стиле интерфейса pandas для вывода графики.

## Знакомство с hvPlot

На рис. 16.7 показана общая архитектура пакета hvPlot. Давайте пройдемся по ней шаг за шагом.

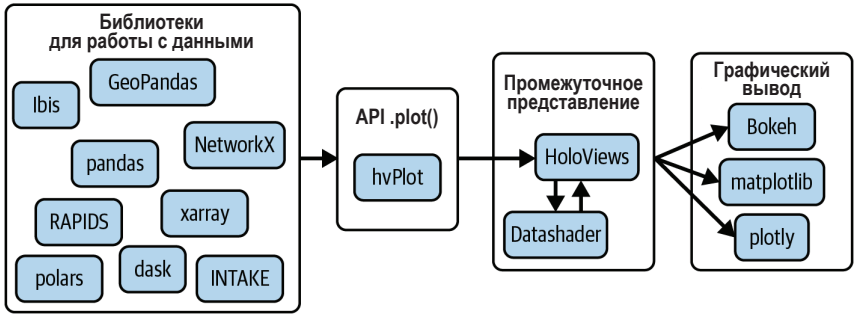



**Рис. 16.7** ❖ hvPlot предлагает универсальный интерфейс для пакетов Bokeh, Matplotlib и Plotly

На вход графические методы пакета hvPlot принимают структуры данных Polars, pandas и других библиотек из экосистемы PyData. Далее hvPlot при помощи библиотеки *HoloViews* строит промежуточное представление для визуализации, зависящее от конкретного пакета. Это промежуточное представление можно воспринимать как план построения визуализации.

При необходимости на этом этапе используется пакет *Datashader* – например, если нужно вывести миллионы точек на карте, что мы сделаем позже.

После этого hvPlot преобразовывает промежуточное представление в спецификацию для пакетов Bokeh, Matplotlib и Plotly. На этой стадии вы можете настраивать внешний вид графиков с помощью синтаксиса выбранного пакета. На заключительном этапе строятся графики, тем самым завершая процесс превращения черно-белых данных в цветные пиксели.

Установить пакет hvPlot можно следующим образом:

```
$ uv pip install hvplot
```

А импортировать – так:

```
import hvplot.polars
```

После импорта пакета hvPlot ваши датафреймы обогатятся за счет нового пространства имен `hvplot`, что позволит вам строить сложные визуализации, не отходя от своего датафрейма.

## Первый график

Начнем с построения диаграммы рассеяния, или точечной диаграммы, которая идеально подходит для сравнения двух непрерывных переменных. Мы воспользуемся теми же датафреймами, которые создали чуть раньше.

Следующий фрагмент кода приведет к построению диаграммы рассеяния с помощью метода `df.hvplot.scatter()`:

```
trips_speed.hvplot.scatter(
    x="distance",
    y="duration",
    color="bike_type", ❶
    xlabel="distance (km)",
    ylabel="duration (h)", ❷
    ylim=(0, 2), ❸
)
```

❶ Первые три аргумента являются самыми важными, поскольку с их помощью мы определяем переменные, которые будут выводиться на осях, а также переменную, уникальные значения которой будут выделяться разными цветами.

❷ Добавлять или изменять существующие подписи осей не обязательно, но это может помочь при чтении графика.

❸ Мы вручную ограничили диапазон значений на вертикальной оси, чтобы очень длительные поездки не мешали визуализации основного подмножества точек. Также вы могли бы исключить такие значения при помощи фильтрации датафрейма.

На рис. 16.8 видно, что электрические велосипеды в среднем быстрее и преодолевают большую дистанцию в сравнении с классическими.

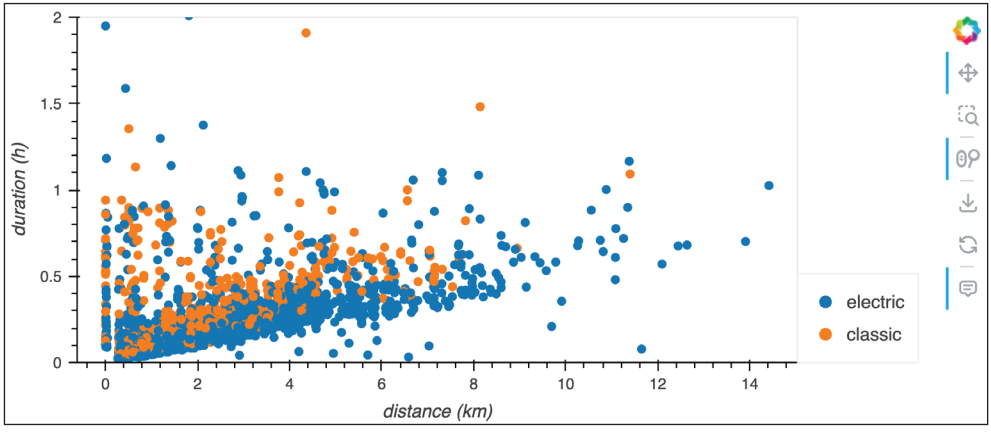



**Рис. 16.8** ❖ Зависимость между расстоянием и длительностью поездки по типам велосипедов

## Методы пространства имен hvPlot

Метод `df.hvplot.scatter()` является одним из многих, доступных в пространстве имен *df.hvplot*. Посмотреть, какие методы есть в этом пространстве, можно при помощи клавиши табуляции:

```
trips.hvplot.<TAB>
```

Среди прочих это пространство имен включает следующие методы:

- *df.hvplot.area()*: выводит диаграмму с областями, отличающуюся от линейного графика цветовым заполнением области под кривой и опциональной возможностью показывать накопление;
- *df.hvplot.bar()*: выводит столбчатую диаграмму с накоплениями или группами;
- *df.hvplot.bivariate()*: выводит двумерный график плотности распределения по набору точек;
- *df.hvplot.box()*: выводит диаграмму размаха (ящики с усами) для сравнения распределений двух или нескольких переменных;
- *df.hvplot.density()*: выводит ядерную оценку плотности для одной или нескольких переменных;
- *df.hvplot.heatmap()*: выводит тепловую карту, помогающую визуализировать переменную по двум независимым измерениям;
- *df.hvplot.hexbins()*: выводит тепловую карту на основе шестиугольников;
- *df.hvplot.hist()*: выводит гистограмму распределения переменной;
- *df.hvplot.line()*: выводит линейный график (как для временных рядов);
- *df.hvplot.scatter()*: выводит диаграмму рассеяния для сравнения двух переменных;

- *df.hvplot.violin()*: выводит скрипичную диаграмму для сравнения распределений переменных с использованием ядерной оценки плотности.

## Pandas в качестве движка

Под капотом пакет hvPlot сначала преобразует датафреймы Polars в формат pandas, копируя при этом только столбцы, необходимые для графика. В большинстве случаев это приемлемо, но не всегда.

Допустим, нам нужно построить тепловую карту, показанную на рис. 16.9. Сначала нам нужно сгруппировать данные по часам и дням, после чего посчитать количество поездок за каждый час:

```
trips_per_day_hour = (
    trips.sort("datetime_start")
    .group_by_dynamic("datetime_start", every="1h")
    .agg(pl.len())
)
```

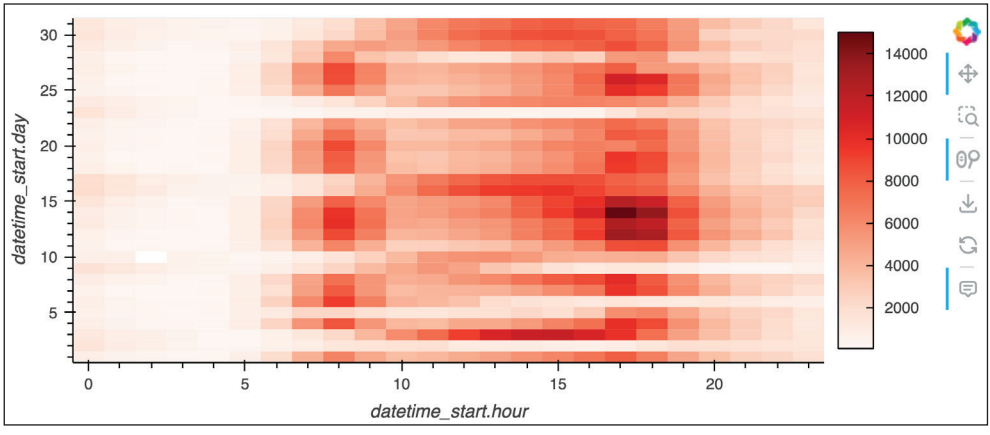



**Рис. 16.9** ❖ Pandas всегда может выручить в ситуациях, когда Polars не справляется

В документации hvPlot упоминаются специальные модификаторы `.hour` и `.day` для извлечения из даты и времени нужных нам компонент. Но, к сожалению, в Polars такие модификаторы пока не поддерживаются, так что следующий код выдаст ошибку:

```
trips_per_day_hour.hvplot.heatmap(
    x="datetime_start.hour", y="datetime_start.day", C="len", cmap="reds"
)
```

Вывод:

```
ValueError: 'datetime_start.day' is not in list
```

Ошибка возникает из-за того, что hvPlot пытается скопировать столбец с именем `datetime_start.day`, который в нашем датафрейме отсутствует. Но ничего страшного, мы всегда при необходимости можем вернуться к Pandas при помощи метода `df.to_pandas()`, как показано ниже:

```
import hvplot.pandas

trips_per_day_hour.to_pandas().hvplot.heatmap(
    x="datetime_start.hour", y="datetime_start.day", C="len", cmap="reds"
)
```

## Ручные преобразования

Теперь давайте построим столбчатую диаграмму. Она бывает полезна для отображения количества элементов в разных группах. Библиотека hvPlot ожидает, что в вашем датафрейме будут находиться уже рассчитанные значения, никаких преобразований за вас она выполнять не будет.

Давайте воспользуемся созданным ранее датафреймом `trips_type_count` для построения столбчатой диаграммы с накоплением:

```
trips_type_counts.hvplot.bar(
    x="rider_type",
    y="len",
    by="bike_type",
    ylabel="count",
    stacked=True,
    color=["orange", "green"],
)
```

На рис. 16.10 видно, что большинство поездок на велосипедах совершают люди, обладающие членским статусом, и чаще всего они берут электрические велосипеды.

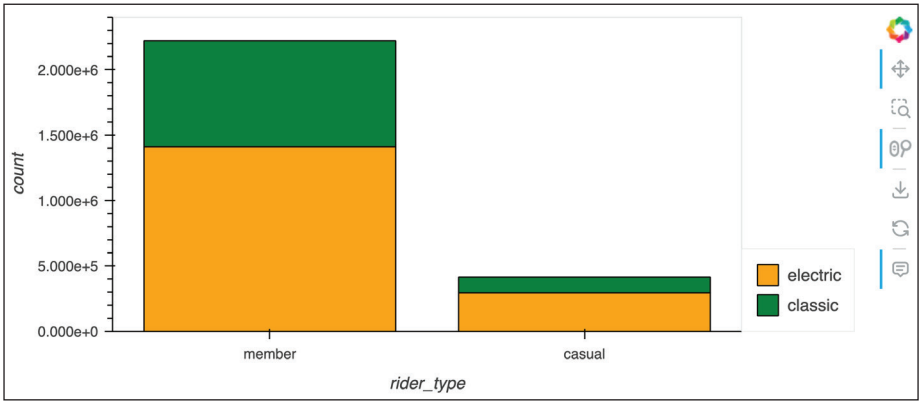



**Рис. 16.10** ❖ Столбчатая диаграмма с накоплением в hvPlot, показывающая количество поездок по типам велосипедов и типам членства

# Изменение движка hvPlot

По умолчанию библиотека hvPlot использует движок *Bokeh*. В большинстве случаев нас это будет устраивать, но иногда бывает полезно сменить движок на Plotly или Matplotlib. К примеру, движок Matplotlib может оказаться полезным, когда мы не испытываем необходимости в интерактивной визуализации или нам нужно соблюсти общий стиль с другими диаграммами.

Изменить движок можно при помощи метода *`hvplot.extension()`*, передав ему строку `"matplotlib"` или `"plotly"`. Для этого нам нужно сначала явно импортировать пакет hvplot, как показано ниже:

```
import hvplot

hvplot.extension("matplotlib")
```

Давайте построим столбчатую диаграмму, как и в предыдущем разделе, но с использованием движка Matplotlib:

```
trips_type_counts.hvplot.bar(
    x="rider_type",
    y="len",
    by="bike_type",
    ylabel="count",
    stacked=True,
    color=["orange", "green"],
)
```

На рис. 16.11 показано, что диаграмма осталась прежней, за исключением нескольких нюансов, характерных для библиотеки Matplotlib.

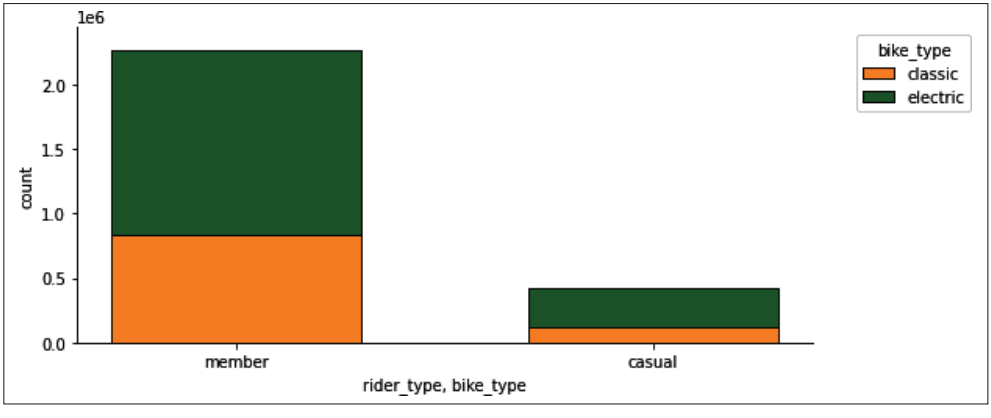



**Рис. 16.11** ❖ Та же диаграмма, но с использованием движка Matplotlib

Вернуть графический движок можно с помощью следующей инструкции:

```
hvplot.extension("bokeh")
```

# Вывод точек данных на географической карте

До сих пор мы не задействовали в визуализациях координаты, присутствующие в нашем датафрейме. Для отображения точек на карте мы можем воспользоваться методом `df.hvplot.points()`, передав параметру `geo` значение `True`. Это позволит обеспечить подобающий вывод точек на карте:

```
trips.hvplot.points(
    x="lon_start",
    y="lat_start",
    datashade=True,
    geo=True,
    tiles="CartoLight",
    width=800,
    height=600,
)
```

На страницах книги это показать, конечно, затруднительно, но карта на рис. 16.12 полностью интерактивна, что позволяет вам увеличивать ее фрагменты и перемещаться по ней. Поскольку мы передали аргументу `datashade` значение `True`, при построении карты оказались задействованы только нужные данные, что способствует эффективности.

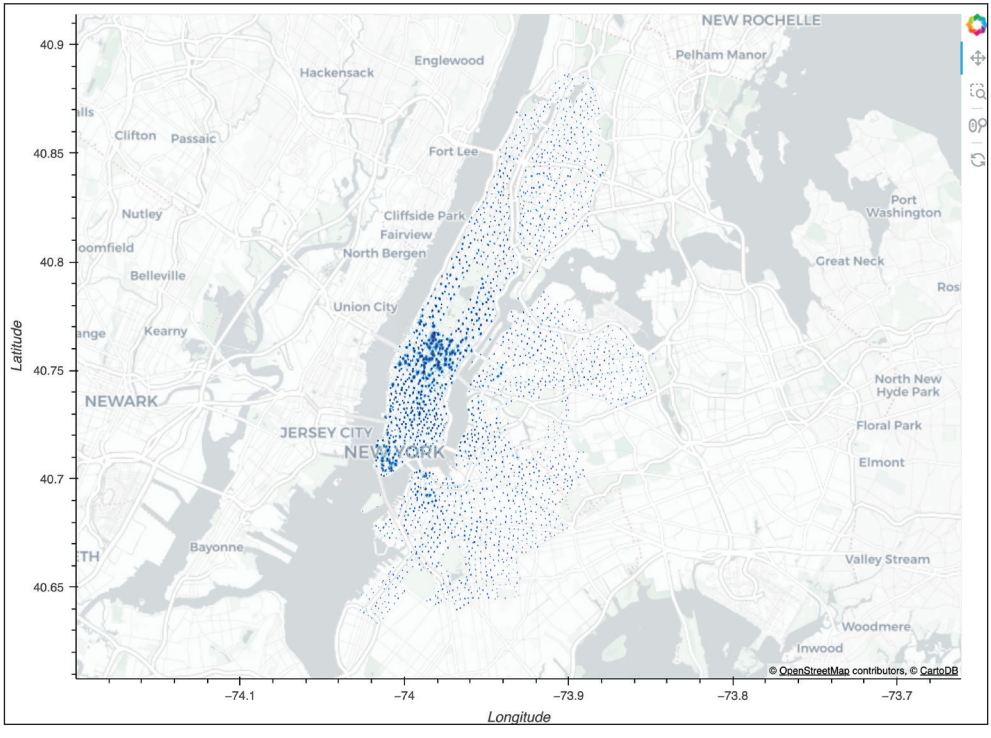



**Рис. 16.12** ❖ Интерактивная географическая карта

# Комбинирование графиков

Иногда одной диаграммы для вывода всей нужной нам информации бывает недостаточно. Библиотека hvPlot позволяет комбинировать множество диаграмм в одной. Существует два типа комбинирования: стекинг и наложение.

Воспользуемся датафреймом `trips_hour_num_speed` для вывода линейного графика. Сначала объединим две диаграммы при помощи оператора +, что позволит разместить графики рядом, как видно на рис. 16.13:

```
(
    trips_hour_num_speed.hvplot.line(x="datetime_start", y="num_trips")
    + trips_hour_num_speed.hvplot.line(x="datetime_start", y="speed")
).cols( ❶
    1
)
```

❶ С помощью метода `cols()` мы обеспечили расположение графиков одного под другим.

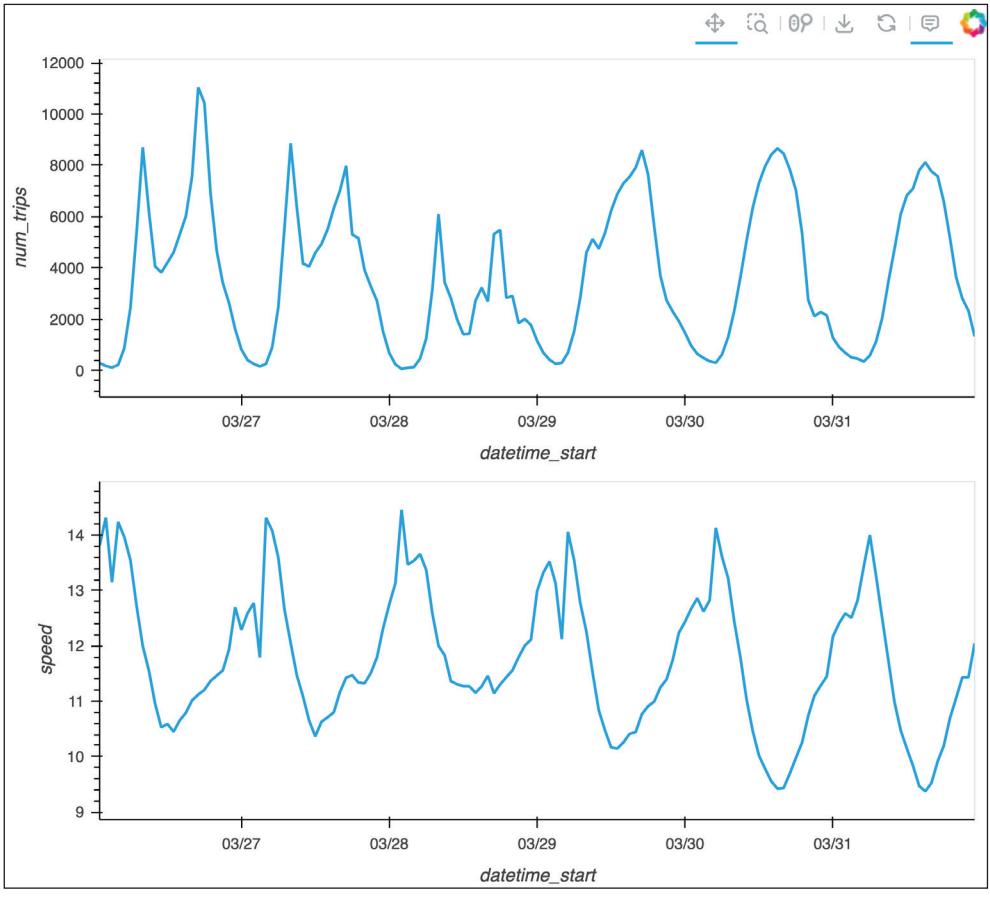



**Рис. 16.13** ❖ Объединение диаграмм в стеке

В следующем фрагменте кода мы воспользуемся оператором *, что позволит нам наложить один график на другой, как показано на рис. 16.14.

```
(
    trips_hour_num_speed.hvplot.line(x="datetime_start", y="num_trips")
    * trips_hour_num_speed.filter(pl.col("num_trips") > 9000).hvplot.scatter(
        x="datetime_start", y="num_trips", c="red", s=50
    )
)
```

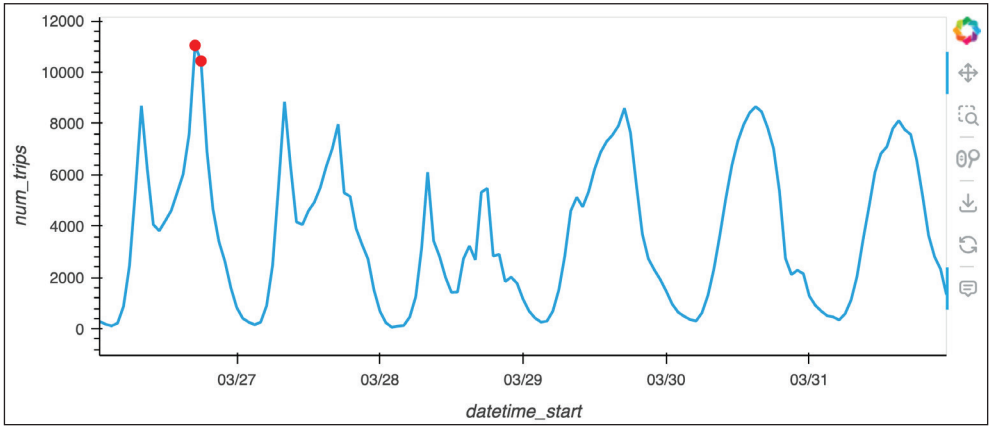



**Рис. 16.14** ❖ Объединение диаграмм с помощью наложения

Обратите внимание, что для наших графиков мы воспользовались разными датафреймами (второй является подмножеством первого) и разными видами диаграмм (линейный график и диаграмма рассеяния).

## Добавление интерактивных виджетов

Движок Bokeh содержит в себе интерактивные инструменты, что позволяет вам перемещаться по графикам, увеличивать их и получать дополнительную информацию при наведении на активные элементы.

С помощью ключевого аргумента `groupby` вы можете добавить на диаграмму один или несколько *виджетов* (widget). Имена колонок, переданные в этом аргументе, делят набор данных на подмножества. С помощью виджетов вы можете выбрать, какие подмножества будут использоваться в визуализации.

Ниже показан пример с группировкой по дате. Тип виджета определяется типом столбца. Как видно на рис. 16.15, виджет для столбца с датой – это слайдер:

```
trips_per_hour = (
    trips.sort("datetime_start")
    .group_by_dynamic("datetime_start", group_by="borough_start", every="1h")
    .agg(pl.len())
```

```
    .with_columns(date=pl.col("datetime_start").dt.date())
)
trips_per_hour
```

Вывод:

```
shape: (2_972, 4)
```

| borough_start | datetime_start | len | date |
|---|---|---|---|
| str | datetime[μs] | u32 | date |
| Manhattan | 2024-03-01 00:00:00 | 480 | 2024-03-01 |
| Manhattan | 2024-03-01 01:00:00 | 294 | 2024-03-01 |
| … | … | … | … |
| Queens | 2024-03-31 22:00:00 | 173 | 2024-03-31 |
| Queens | 2024-03-31 23:00:00 | 126 | 2024-03-31 |

```
trips_per_hour.hvplot.line(
    x="datetime_start",
    by="borough_start",
    groupby="date",
    widget_location="left_top",
)
```

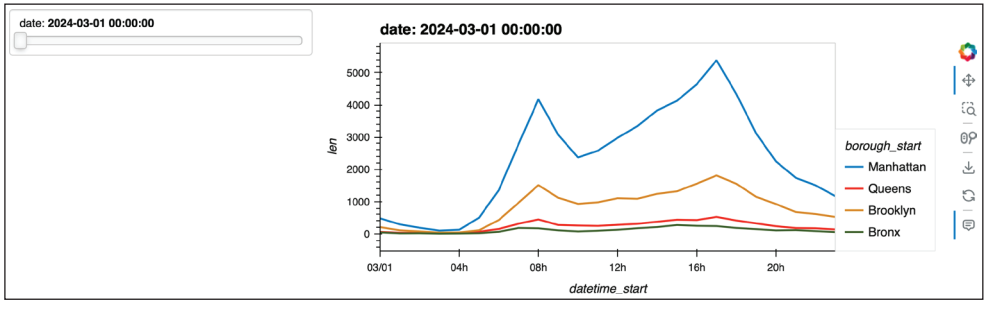



**Рис. 16.15** ❖ Интерактивные виджеты на диаграмму можно добавить с помощью ключевого аргумента `groupby`

Вы также можете передать этому аргументу список имен столбцов для создания нескольких виджетов.

# Готовые к публикации графики при помощи plotnine

*Plotnine* (*https://plotnine.org*) представляет собой пакет для визуализации данных, созданный Хассаном Кибириджем (Hassan Kibirige). API этого пакета

напоминает интерфейс ggplot2 – известного пакета в языке R, созданного Хэдли Уикемом и другими разработчиками.

Лично мы при построении графиков и диаграмм отдаем предпочтение именно библиотеке plotnine. И в этой книге мы по большей части пользовались именно этим пакетом для визуализации.

## Знакомство с plotnine

В основе библиотеки plotnine лежит принцип, называемый *многослойной грамматикой графики* (layered grammar of graphics)[1]. Этот принцип в библиотеке реализован совместно с интерфейсом, позволяющим быстро и интерактивно создавать самые разные типы визуализаций почти без обращения к документации.

Установить пакет plotnine можно следующим образом:

```
$ uv pip install 'plotnine[all]'
```

А его импорт осуществляется так:

```
from plotnine import *
```

**Импорт всего на свете**

Хотя обычно импорт всего содержимого пакета в глобальное пространство имен не приветствуется, мы позволили себе сделать это здесь, поскольку это даст нам возможность более удобно вызывать функции plotnine. Если вы не хотите засорять свое глобальное окружение, можете воспользоваться инструкцией `import plotnine as p9` и к каждой функции добавлять префикс `p9`.

## Диаграммы для исследования данных

Библиотека plotnine позволяет создавать диаграммы на лету, используя при этом относительно немного кода. Давайте начнем с диаграммы рассеяния:

```
(
    ggplot(trips_speed, aes(x="distance", y="duration", color="bike_type"))
    + geom_point() ❶
)
```

❶ В plotnine функции создания визуализаций объединяются при помощи оператора `+`.

Вывод графика показан на рис. 16.16.

[1] См. статью Хэдли Уикема A Layered Grammar of Graphics: *http://vita.had.co.nz/papers/layered-grammar.pdf*.

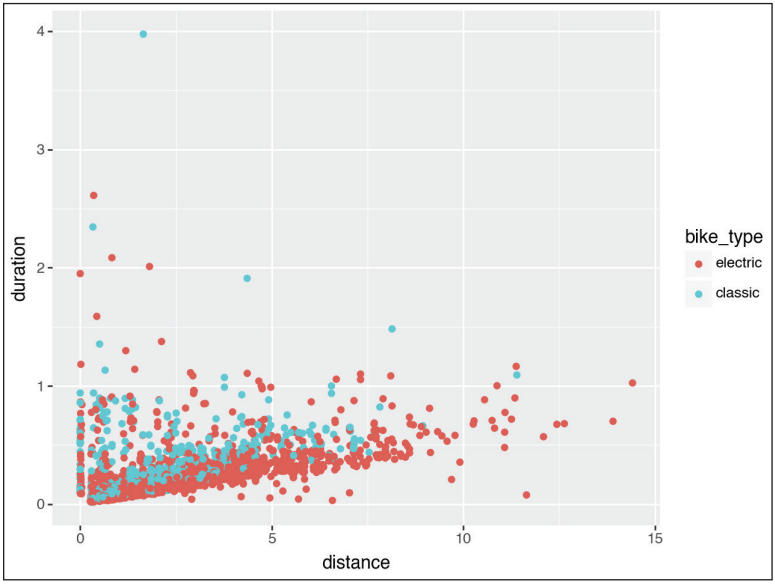



**Рис. 16.16** ❖ Диаграмма рассеяния, построенная при помощи пакета plotnine

В этом примере отражена сама суть библиотеки plotnine. С помощью функции *aes()* столбцы в датафрейме сопоставляются с так называемыми эстетиками в визуализации. Эстетики представляют собой некие свойства используемой геометрии, в нашем случае это точки на карте, посредством функции *geom_point()*.

Некоторые геометрии предварительно выполняют статистические преобразования. К примеру, функция `geom_hist()` в показанном ниже фрагменте кода создает некоторое количество корзин для всего диапазона данных и рассчитывает количество поездок, принадлежащих той или иной корзине (см. рис. 16.17). Эти количества используются в эстетике `y`:

```
ggplot(trips_speed, aes(x="distance")) + geom_histogram()
```

Функция *geom_bar()*, используемая для построения столбчатой диаграммы, также выполняет статистические преобразования. Поскольку мы указали цвет заполнения при помощи эстетики `fill`, plotnine автоматически вывел столбчатую диаграмму с накоплением, как видно на рис. 16.18:

```
ggplot(trips, aes(x="rider_type", fill="bike_type")) + geom_bar()
```

Эстетики также могут содержать статичные значения. В этом случае аргумент не должен находиться внутри функции `aes()`, как показано ниже. Вывод графика – на рис. 16.19.

```
ggplot(trips_speed, aes(x="distance", fill="bike_type")) + geom_density(
    alpha=0.7, color="none"
)
```

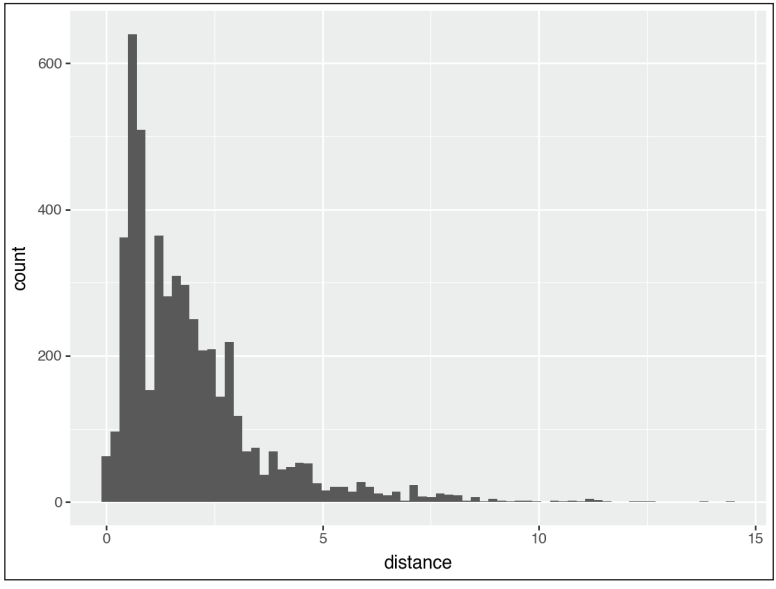



**Рис. 16.17** ❖ Гистограмма, построенная при помощи пакета plotnine

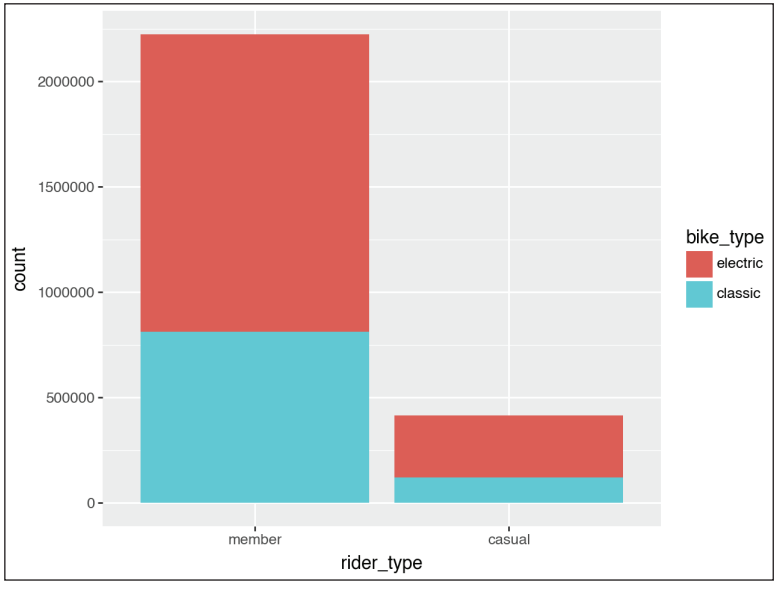



**Рис. 16.18** ❖ Столбчатая диаграмма, построенная при помощи пакета plotnine

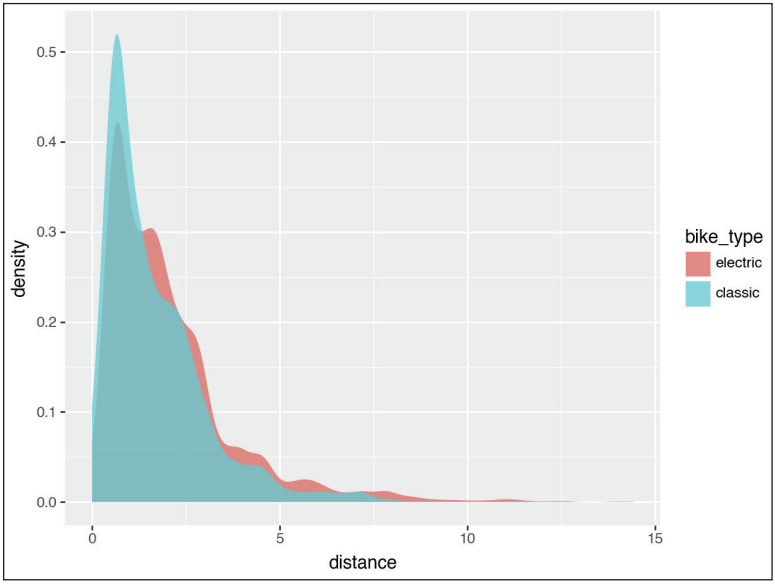



**Рис. 16.19** ❖ График плотности распределения, построенный при помощи пакета plotnine

Каждая функция геометрии добавляет на визуализацию новый слой – именно так в библиотеке plotnine реализована поддержка многослойных графиков. Давайте воссоздадим при помощи этого пакета линейный график, который ранее создали посредством пакета hvPlot. Вывод показан на рис. 16.20:

```
(
    ggplot(trips_hour_num_speed, aes(x="datetime_start", y="num_trips"))
    + geom_line(size=1, color="steelblue")
    + geom_point(
        data=trips_hour_num_speed.filter(pl.col("num_trips") > 9000),
        color="red",
        size=4,
    )
)
```

Обратите внимание, что нам понадобилось указать привязку к эстетикам лишь раз, и во всех слоях она используется по умолчанию. Стоит также упомянуть, что привязки и сами датафреймы в каждом слое могут быть свои.

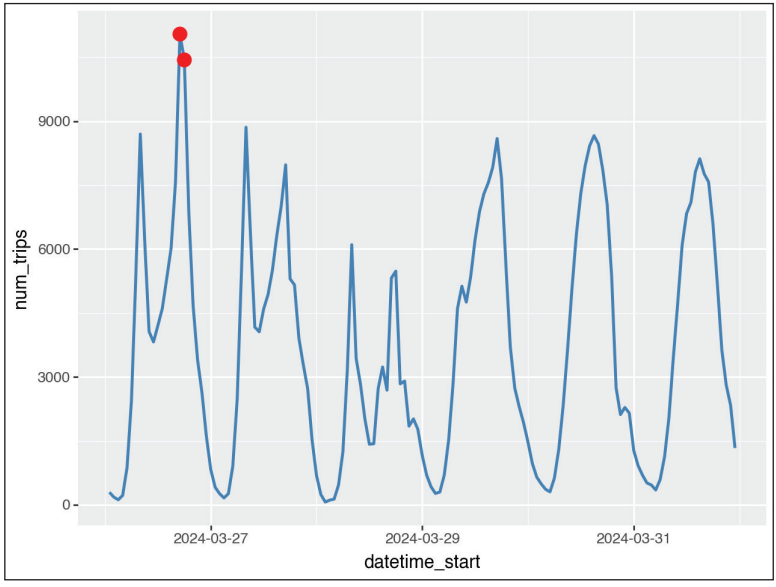



**Рис. 16.20** ❖ Многослойный график, построенный при помощи пакета plotnine

# Графики для публикации

До этого момента мы создавали свои визуализации с помощью всего нескольких строк кода, что типично для разведочного анализа данных. Такие графики могут выглядеть не идеально, но для целей исследования они вполне годятся. Когда мы готовы поделиться своей работой с коллегами, мы обычно украшаем графики, добавляя на них цвета, подписи и нужные шрифты.

Давайте попробуем создать более изысканную диаграмму рассеяния. Сначала мы добавим дополнительный слой, а затем разделим наш график на четыре панели для удобства.

В следующем фрагменте кода мы подготавливаем наш датафрейм для сравнения переменных, соответствующих расстоянию и длительности поездок. На этот раз мы будем анализировать станции велосипедов с расчетом медианного расстояния и длительности поездок для каждой станции. Цель состоит в определении корреляции между расстояниями и длительностью поездок в рамках одного округа:

```
trips_speed = (
    trips.group_by("neighborhood_start", "neighborhood_end")
    .agg(
        pl.col("duration").dt.total_seconds().median() / 60,
        pl.col("distance").median(),
        pl.col("borough_start").first(),
        pl.col("borough_end").first(),
        pl.len(),
```

```
    )
    .filter(
        (pl.col("len") > 30)
        & (pl.col("distance") > 0.2)
        & (pl.col("neighborhood_start") != pl.col("neighborhood_end")),
    )
    .with_columns(speed=pl.col("distance") / pl.col("duration"))
    .sort("borough_start")
)
trips_speed
```

Вывод:

```
shape: (2_962, 8)
```

| neighborhood_start | neighborhood_end | duration | … | borough_end | len | speed |
|---|---|---|---|---|---|---|
| str | str | f64 | | str | u32 | f64 |
| Longwood | Mott Haven | 6.35 | … | Bronx | 1317 | 0.17757 |
| Highbridge | Claremont Village | 8.483333 | … | Bronx | 163 | 0.155674 |
| … | … | … | … | … | … | … |
| Astoria | Corona | 21.233333 | … | Queens | 51 | 0.233306 |
| Elmhurst | Williamsburg | 27.133333 | … | Brooklyn | 42 | 0.24442 |

Ниже показан код для создания первой диаграммы рассеяния. Каждая строка (а на графике, показанном на рис. 16.21, каждая точка) содержит пару из начальной и конечной станций из одного округа:

```
(
    ggplot(
        data=trips_speed.filter(
            pl.col("borough_start") == pl.col("borough_end")
        ),
        mapping=aes(x="distance", y="duration", color="borough_end"),
    )
    + geom_point(size=0.25, alpha=0.5)
    + geom_smooth(method="lowess", size=2, se=False, alpha=0.8) ❶
    + xlim(0, 15)
    + ylim(0, 60)
    + scale_color_brewer(type="qualitative", palette="Set1") ❷
    + labs(
        title="Расстояния и длительности поездок в рамках одного округа",
        x="Расстояние (км)",
        y="Длительность (мин)",
        color="Округ",
    )
    + theme_tufte(base_family="Guardian Sans", base_size=14) ❸
    + theme(
        figure_size=(8, 6),
```

```
        dpi=300,
        plot_background=element_rect(color="#ffffff"),
    )
)
```

❶ Добавляем линию регрессии.
❷ Изменяем цветовую палитру для точек и линий.
❸ Используем тему, навеянную минималистическими принципами визуализации признанного специалиста в области информационного дизайна Эдварда Тафти (Edward Tufte).

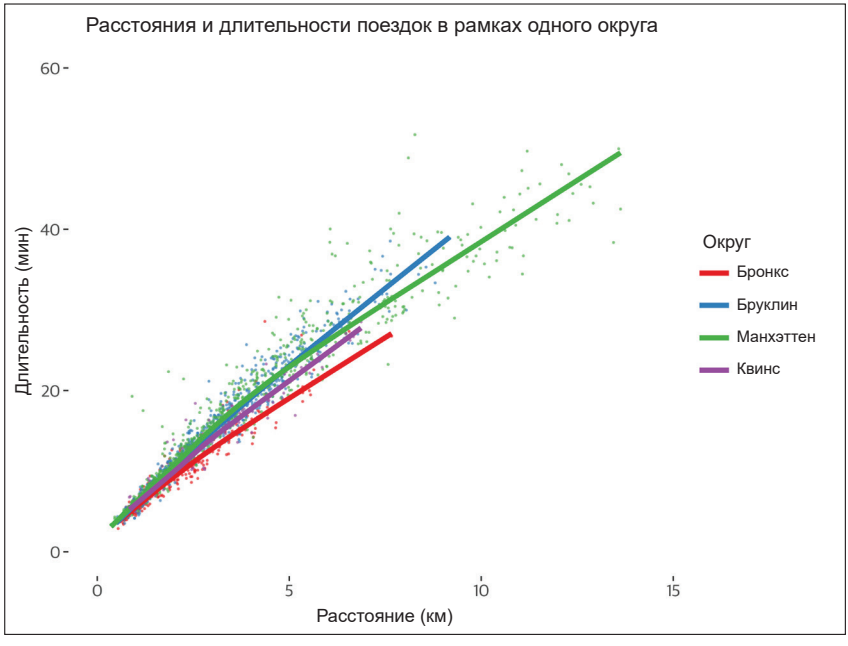



**Рис. 16.21** ❖ Расстояния и длительности поездок в рамках одного округа

Как видите, этот фрагмент кода оказался гораздо более многословным в сравнении с предыдущими. Причина в том, что мы вынуждены были изменить многие параметры, принятые в библиотеке plotnine по умолчанию. Как правило, чем больше изменений вы хотите внести в визуализацию, тем больше кода вам придется написать.

Если мы обратим примененный к датафрейму фильтр, то сможем рассмотреть поездки, которые начинались и заканчивались в разных округах Нью-Йорка. Поскольку у нас есть четыре округа, из которых могут начинаться поездки, логично будет создать четыре диаграммы рассеяния. Воспользуемся функцией *facet_wrap()* для создания четырех панелей в визуализации, по одной для каждого округа:

```
(
    ggplot(
```

```
        data=trips_speed.filter(
            pl.col("borough_start") != pl.col("borough_end")
        ).with_columns(
            ("From " + pl.col("borough_start")).alias("borough_start")
        ),
        mapping=aes(x="distance", y="duration", color="borough_end"),
    )
    + geom_point(size=0.25, alpha=0.5)
    + geom_smooth(method="lowess", size=2, se=False, alpha=0.8)
    + xlim(0, 15)
    + ylim(0, 60)
    + scale_color_brewer(type="qualitative", palette="Set1")
    + facet_wrap("borough_start")
    + labs(
        title="Расстояния и длительности поездок между округами",
        x="Расстояние (км)",
        y="Длительность (мин)",
        color="В округ",
    )
    + theme_linedraw(base_family="Guardian Sans", base_size=14)
    + theme(figure_size=(8, 6), dpi=300)
)
```

Результат показан на рис. 16.22.

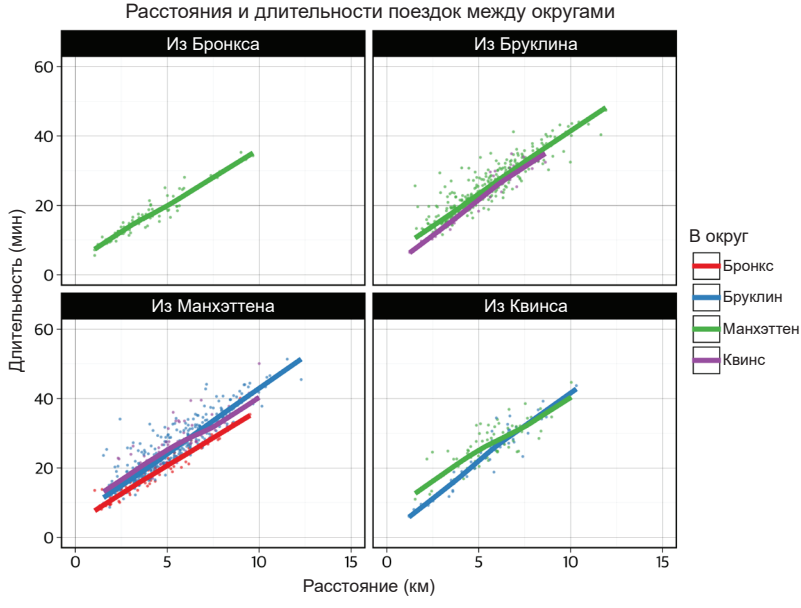



**Рис. 16.22** ❖ Расстояния и длительности поездок между округами

Если вы сравните два последних фрагмента кода, то обнаружите в них очень много общего. Мы только изменили аргумент `data` в функции `ggplot()`,

добавили функцию `facet_wrap()` во втором фрагменте для создания четырех панелей и обновили пару подписей. Все это возможно благодаря очень удобному и органичному интерфейсу plotnine. Визуализации здесь создаются последовательно по цепочке, а не путем добавления ключевых аргументов в один и тот же метод.

Подробнее о библиотеке plotnine можно почитать на ее официальном сайте по адресу *https://plotnine.org* и в посте Йеруна в блоге по адресу *https://jeroenjanssens.com/plotnine*.

# Стилизация датафреймов при помощи Great Tables

До сих пор мы обсуждали два варианта представления данных: при помощи датафреймов в табличном виде и посредством визуализаций – в графическом.

Но есть и третий способ, занимающий промежуточное положение. Мы говорим о таблицах. Пакет *Great Tables* (*https://posit-dev.github.io/great-tables/articles/intro.html*) от Ричарда Янноне (Rich Iannone) и Майкла Чоу (Michael Chow) позволяет создавать поистине восхитительные таблицы!

Таблица бывает полезна, когда в ней представлены данные в чистом и структурированном виде. Такой вид включает в себя:

- хорошо различимые и читабельные имена столбцов;
- числовые значения с надлежащим форматированием;
- группировку строк;
- стилизацию с целью привлечения внимания к важным значениям;
- аннотации, такие как заголовки, метки и сноски.

Философия пакета Great Tables базируется на наборе связанных табличных компонентов, показанных на рис. 16.23. Начиная с датафрейма на входе вы можете последовательно сцеплять методы для добавления элементов и применения форматирования.

Установить пакет Great Tables можно следующим образом:

```
$ uv pip install great_tables
```

Давайте подготовим наш датафрейм к отображению в виде новой таблицы. В следующем фрагменте кода определяются три самые загруженные станции в округах:

```
import polars.selectors as cs

busiest_stations = (
    trips.group_by( ❶
```

```
        station=pl.col("station_start"), date=pl.col("datetime_start").dt.date()
    )
    .agg(
        borough=pl.col("borough_start").first(),
        neighborhood=pl.col("neighborhood_start").first(),
        num_rides=pl.len(),
        percent_member=(pl.col("rider_type") == "member").mean(),
        percent_electric=(pl.col("bike_type") == "electric").mean(),
    )
    .sort("date")
    .group_by("station")
    .agg(
        cs.string().first(),
        cs.numeric().mean(),
        pl.col("num_rides").alias("rides_per_day"), ❷
    )
    .sort("num_rides", descending=True)
    .group_by("borough", maintain_order=True)
    .head(3)
)

busiest_stations
```

❶ Эта первая агрегация нужна, потому что мы хотим отобразить количество поездок для каждой станции по дням в виде встроенных графиков (увидим это позже).
❷ Значения в этом столбце понадобятся нам при построении встроенных графиков.


**Рис. 16.23** ❖ Компоненты таблицы в Great Tables (показаны с разрешения авторов пакета)

Вывод:

```
shape: (12, 7)
```

| borough | station | neighborhood | num_rides | percent_member | percent_electric | rides_per_day |
|---|---|---|---|---|---|---|
| str | str | str | f64 | f64 | f64 | list[u32] |
| Manhattan | W 21 St & 6 Ave | Chelsea | 354.16129 | 0.913765 | 0.583606 | [325, 88, … 306] |
| … | … | … | … | … | … | … |
| Bronx | Plaza Dr & W 170 St | Mount Eden | 31.709677 | 0.837427 | 0.948925 | [30, 19, … 33] |

Есть два способа стилизации датафрейма в формате Great Tables. Первый состоит в использовании атрибута датафрейма *style*, встроенного в Polars, как показано ниже:

```
df.style
```

Второй заключается в применении функции *GT()* из пакета Great Tables. Для этого необходимо сначала импортировать эту функцию следующим образом:

```
from great_tables import GT

GT(df)
```

В результате мы получим объект *GT*, который в дальнейшем можно использовать для настройки внешнего вида вашей таблицы. На момент написания книги атрибут датафрейма `style` был помечен в Polars как нестабильный, так что мы воспользуемся вторым из предложенных вариантов применительно к датафрейму `busiest_stations`. Результат показан на рис. 16.24:

```
GT(busiest_stations)
```

Конечно, из коробки таблица выглядит не лучшим образом, в особенности из-за очень широкой последней колонки. Так что здесь есть над чем поработать. В следующем фрагменте кода мы стилизуем нашу таблицу и добавим ей структуры с помощью разных методов:

```
from great_tables import style, md

(
    GT(busiest_stations)
    .tab_stub(rowname_col="station", groupname_col="borough") ❶
```

```
    .cols_label( ❷
        neighborhood="Neighborhood",
        num_rides="Mean Daily Rides",
        percent_member="Members",
        percent_electric="E-Bikes",
        rides_per_day="Rides Per Day",
    )
    .tab_header(
        title="Busiest Bike Stations in NYC",
        subtitle="In March 2024, Per Borough",
    )
    .tab_stubhead(label="Station")
    .fmt_number(columns="num_rides", decimals=1)
    .fmt_percent(columns=cs.starts_with("percent_"), decimals=0) ❸
    .fmt_nanoplot(columns="rides_per_day", reference_line="mean")
    .data_color(columns="num_rides", palette="Blues")
    .tab_options(row_group_font_weight="bold")
    .tab_source_note(
        source_note=md(
            "Source: [NYC Citi Bike](https://citibikenyc.com/system-data)"
        )
    )
)
```

❶ Группируем станции по округам для структуры.
❷ Мы можем давать колонкам осмысленные имена без необходимости вносить изменения в датафрейм.
❸ Пакет Great Tables позволяет передавать в виде аргументов селекторы столбцов, что делает код более компактным.

| borough | station | neighborhood | num_rides | percent_member | percent_electric | rides_per_day |
|---|---|---|---|---|---|---|
| Manhattan | W 21 St & 6 Ave | Chelsea | 354.16129032258067 | 0.913764948841179 | 0.5836060628339382 | [325, 88, 409, 433, 221, 219, 365, 369, 162, 307, 378, 502, 507, 498, 447, 415, 361, 385, 385, 361, 369, 366, 91, 370, 446, 480, 407, 210, 428, 369, 306] |
| Manhattan | Broadway & W 58 St | Midtown | 307.5483870967742 | 0.7963690113856159 | 0.7057834322293212 | [222, 65, 446, 299, 122, 154, 340, 332, 122, 171, 278, 373, 359, 433, 660, 766, 662, 267, 281, 284, 243, 237, 41, 210, 279, 304, 280, 172, 407, 360, 365] |
| Manhattan | 8 Ave & W 31 St | Chelsea | 288.258064516129 | 0.8653562450153743 | 0.6404165834717456 | [279, 86, 245, 360, 166, 245, 322, 317, 92, 153, 317, 449, 421, 497, 406, 274, 277, 271, 395, 318, 350, 282, 62, 176, 347, 404, 336, 217, 344, 242, 286] |
| Brooklyn | Metropolitan Ave & Bedford Ave | Williamsburg | 184.8709677419355 | 0.8548570633632913 | 0.6796751094803254 | [173, 59, 231, 208, 114, 97, 216, 240, 88, 134, 172, 244, 247, 200, 219, 225, 269, 169, 158, 173, 176, 181, 88, 218, 211, 230, 180, 90, 224, 256, 241] |
| Brooklyn | N 7 St & Driggs Ave | Williamsburg | 145.70967741935485 | 0.8600288134761424 | 0.655533737720296 | [134, 51, 183, 117, 98, 86, 149, 176, 71, 99, 142, 168, 191, 202, 183, 235, 171, 145, 172, 183, 152, 162, 35, 98, 151, 189, 172, 107, 169, 179, 147] |
| Brooklyn | Hanson Pl & Ashland Pl | Fort Greene | 143.83870967741936 | 0.8361605984376713 | 0.6139227831846854 | [145, 46, 133, 158, 105, 111, 148, 178, 50, 86, 163, 155, 191, 224, 230, 171, 161, 155, 177, 152, 148, 156, 29, 103, 150, 186, 165, 108, 168, 149, 158] |
| Queens | Queens Plaza North & Crescent St | Long Island City | 126.74193548387096 | 0.8537365542705813 | 0.5707317493983467 | [89, 34, 188, 160, 55, 61, 118, 138, 80, 115, 135, 195, 181, 203, 196, 220, 167, 133, 113, 100, 104, 103, 28, 91, 137, 163, 109, 71, 150, 159, 133] |
| Queens | Vernon Blvd & 50 Ave | Long Island City | 95.16129032258064 | 0.8903794246434368 | 0.6630370994599845 | [94, 28, 96, 102, 56, 58, 100, 95, 26, 34, 85, 120, 124, 124, 102, 46, 56, 109, 112, 88, 115, 132, 39, 103, 155, 152, 121, 113, 121, 122, 122] |
| Queens | 31 St & Newtown Ave | Astoria | 78.16129032258064 | 0.8830932418797895 | 0.5968191831951197 | [77, 29, 80, 96, 43, 57, 84, 88, 36, 54, 98, 110, 103, 109, 107, 62, 65, 112, 93, 92, 79, 86, 34, 70, 90, 93, 95, 53, 83, 84, 61] |
| Bronx | Melrose Ave & E 150 St | Melrose | 41.61290322580645 | 0.8341575787497405 | 0.8946989054437855 | [43, 15, 36, 54, 39, 24, 63, 58, 24, 32, 38, 45, 68, 59, 73, 37, 39, 47, 43, 41, 45, 41, 10, 29, 43, 49, 47, 34, 48, 28, 38] |
| Bronx | E 161 St & River Ave | Concourse | 35.48387096774193 | 0.7444040715853226 | 0.8791680897318485 | [32, 12, 69, 53, 17, 15, 35, 23, 8, 22, 20, 45, 52, 69, 44, 72, 52, 25, 35, 35, 24, 21, 6, 32, 35, 38, 32, 18, 37, 72, 50] |
| Bronx | Plaza Dr & W 170 St | Mount Eden | 31.70967741935484 | 0.8374265559706529 | 0.948924783664995 | [30, 19, 22, 23, 13, 23, 36, 41, 22, 23, 29, 27, 41, 38, 37, 32, 39, 44, 37, 33, 38, 34, 12, 28, 42, 36, 42, 30, 46, 33, 33] |

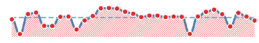
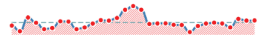
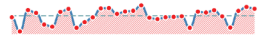
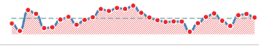
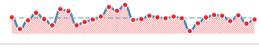
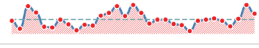
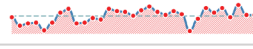
**Рис. 16.24** ❖ Таблица, созданная с помощью пакета Great Tables

Итоговая таблица показана на рис. 16.25. В последнем столбце таблицы выведены так называемые встроенные графики, или нанографики. С помощью них мы отображаем количество поездок по дням в разрезе станций.

Бывает очень удобно быстро построить начальную таблицу с помощью пакета Great Tables, а затем итеративно доводить ее до требуемого вида.

Busiest Bike Stations in NYC

In March 2024, Per Borough

| Station | Neighborhood | Mean Daily Rides | Members | E-Bikes | Rides Per Day |
|---|---|---|---|---|---|
| **Manhattan** | | | | | |
| W 21 St & 6 Ave | Chelsea | 354.2 | 91% | 58% |  |
| Broadway & W 58 St | Midtown | 307.5 | 80% | 71% |  |
| 8 Ave & W 31 St | Chelsea | 288.3 | 87% | 64% |  |
| **Brooklyn** | | | | | |
| Metropolitan Ave & Bedford Ave | Williamsburg | 184.9 | 85% | 68% |  |
| N 7 St & Driggs Ave | Williamsburg | 145.7 | 86% | 66% |  |
| Hanson Pl & Ashland Pl | Fort Greene | 143.8 | 84% | 61% |  |
| **Queens** | | | | | |
| Queens Plaza North & Crescent St | Long Island City | 126.7 | 85% | 57% |  |
| Vernon Blvd & 50 Ave | Long Island City | 95.2 | 89% | 66% |  |
| 31 St & Newtown Ave | Astoria | 78.2 | 88% | 60% |  |
| **Bronx** | | | | | |
| Melrose Ave & E 150 St | Melrose | 41.6 | 83% | 89% |  |
| E 161 St & River Ave | Concourse | 35.5 | 74% | 88% |  |
| Plaza Dr & W 170 St | Mount Eden | 31.7 | 84% | 95% |  |

Source: NYC Citi Bike

**Рис. 16.25** ❖ Таблица с информацией о трех наиболее загруженных станциях в каждом округе

# Заключение

В этой главе мы рассмотрели разные способы визуализации датафреймов при помощи графиков и превращения их в структурированные и красочно оформленные таблицы.

Вы узнали, что:

- существует множество пакетов для визуализации данных;
- в Polars есть встроенные инструменты для визуализации данных, в основе которых лежит библиотека Altair;
- библиотека hvPlot может использовать в качестве движков пакеты Bokeh, Matplotlib и Plotly;

- вы всегда можете воспользоваться экспортом в Pandas, если нужный вам пакет визуализации данных не поддерживает Polars;
- plotnine – это мощный пакет для визуализации данных, в основе которого лежат принципы грамматики графики, позволяющий создавать как быстрые графики и диаграммы на лету для разведочного анализа данных, так и полноценные визуализации для презентаций;
- таблица может быть информативнее любого графика, если стилизовать и структурировать ее при помощи, например, пакета Great Tables.

В следующей главе мы узнаем о способах расширения возможностей библиотеки Polars.

# Глава 17

# Расширения Polars

Как вы убедились при чтении предыдущих глав книги, API Polars предлагает очень богатые возможности и покрывает большую часть требующегося функционала. Но бывают случаи, когда вам может не хватить встроенного функционала Polars. Это может происходить при выполнении нестандартных задач, а также при желании еще больше оптимизировать свой код.

В этой главе вы узнаете:

- как применять пользовательские функции, написанные на Python, к данным в Polars;
- как регистрировать пространства имен в Polars;
- как писать плагины на Rust и запускать их на движке Polars для извлечения максимальной эффективности;
- как использовать в этих плагинах крейты Rust.

Все инструкции по загрузке нужных файлов находятся в главе 2. Мы предполагаем, что все рабочие файлы вы поместили в директорию *data*.

## Пользовательские функции на Python

В Polars присутствует расширенный набор выражений, позволяющих выполнять самые разные операции. Но время от времени вам может быть необходимо выполнить операцию, для которой нет соответствующего выражения в Polars или которая должна выполняться при помощи внешнего пакета. В Polars для этих целей реализована возможность использования *пользовательских функций* (user-defined function – UDF). Методы Polars, позволяющие это делать:

- `Expr.map_elements()`: применяет функцию, написанную на Python, к каждому элементу объекта Series;
- `Expr.map_batches()`: применяет функцию на Python к объекту Series или последовательности объектов Series;
- `Expr.map_groups()`: применяет функцию на Python к каждой группе в контексте GroupBy;

- `Expr.pipe()`: применяет функцию на Python к выражению;
- `df.pipe()`: применяет функцию на Python ко всему датафрейму (и ленивому датафрейму).

Давайте посмотрим, как можно использовать эти методы для применения функций, написанных на Python, к вашим данным.

## Применение функций к элементам

Метод *`Expr.map_elements()`* позволяет применять функции, написанные на языке Python, к каждому элементу объекта Series по отдельности, если вам ничего не нужно знать о других элементах Series.

В примере, показанном ниже, применяется анализ тональности текста к столбцу с отзывами в датафрейме:

```
from textblob import TextBlob

def analyze_sentiment(review):
    return TextBlob(review).sentiment.polarity

reviews = pl.DataFrame(
    {
        "reviews": [
            "This product is great!",
            "Terrible service.",
            "Okay, but not what I expected.",
            "Excellent! I love it.",
        ]
    }
)

reviews.with_columns(
    sentiment_score=pl.col("reviews").map_elements(
        analyze_sentiment, return_dtype=pl.Float64
    )
)
```

Вывод:

```
shape: (4, 2)
┌─────────────────────────────────┬─────────────────┐
│ reviews                         ┆ sentiment_score │
│ ---                             ┆ ---             │
│ str                             ┆ f64             │
╞═════════════════════════════════╪═════════════════╡
│ This product is great!          ┆ 1.0             │
│ Terrible service.               ┆ -1.0            │
│ Okay, but not what I expected.  ┆ 0.2             │
│ Excellent! I love it.           ┆ 0.75            │
└─────────────────────────────────┴─────────────────┘
```

В этом примере мы воспользовались методом `Expr.map_elements()` для применения функции `analyze_sentiment()` к каждому элементу объекта Series, представленного столбцом `reviews`. В результате в датафрейм был добавлен столбец с именем `sentiment_score`, значения в котором соответствуют диапазону от –1,0 (крайне отрицательная тональность отзыва) до 1,0 (крайне положительная тональность отзыва). Значение 0,0 соответствует нейтральному отзыву.

### Предупреждения, связанные с неэффективными операциями

При использовании функций, реализованных на языке Python, в Polars всегда важно помнить о том, что они не смогут обеспечить такое же быстродействие, как родные функции Polars. Причина в том, что операции в Polars по большей части используют низкоуровневый язык Rust. В то же время при запуске функции на Python происходит следующее:

- функция запускает более медленный байт-код Python вместо более быстрого байт-кода Rust;
- функции на Python имеют ограничение в виде *глобальной блокировки интерпретатора* (Global Interpreter Lock – GIL)[1], не позволяющее выполнять операции параллельно. Особенно губительно это может быть при использовании операций `df.group_by()`, в которых функции агрегации обычно вызываются для каждой группы параллельно.

Применение пользовательских или анонимных функций, написанных на Polars, к датафреймам в Polars необходимо рассматривать как крайнюю меру. Если вы видите в Polars предупреждение *`PolarsInefficientMapWarning`*, это означает, что с большой долей вероятности можно найти альтернативный способ с использованием выражений Polars. Применять функции на Python можно только в том случае, если вы перебрали все возможные способы решения задачи при помощи родных выражений Polars или их комбинаций.

Запуск приведенного ниже фрагмента кода приведет к появлению предупреждения `PolarsInefficientMapWarning` по причине применения тривиальной функции к объекту Series:

```
ints = pl.DataFrame({"x": [1, 2, 3, 4]})

def add_one(x):
    return x + 1

ints.with_columns(
    pl.col("x")
    .map_elements(
        add_one,
        return_dtype=pl.Int64,
    )
    .alias("x + 1")
)
```

[1] Новые версии Python будут содержать интерпретаторы, не подверженные ограничению GIL, но это не значит, что они автоматически будут задействовать в работе все доступные ядра процессора.

Вывод:

```
PolarsInefficientMapWarning:
Expr.map_elements is significantly slower than the native expressions API.
Only use if you absolutely CANNOT implement your logic otherwise.
Replace this expression...
  - pl.col("x").map_elements(add_one)
with this one instead:
  + pl.col("x") + 1

  .map_elements(

shape: (4, 2)
┌─────┬─────────┐
│ x   ┆ x + 1   │
│ --- ┆ ---     │
│ i64 ┆ i64     │
╞═════╪═════════╡
│ 1   ┆ 2       │
│ 2   ┆ 3       │
│ 3   ┆ 4       │
│ 4   ┆ 5       │
└─────┴─────────┘
```

Всегда читайте текст предупреждений и используйте полученную информацию для оптимизации своей логики.

# Применение функций к объектам Series

Метод *`Expr.map_batches()`* позволяет применять функцию на Python к объекту Series или последовательности объектов Series. Он может быть исключительно полезным, когда вам требуется знать что-то о других элементах объекта Series или при необходимости применить функцию к нескольким объектам Series одновременно. В табл. 17.1 показаны аргументы метода `Expr.map_batches()`.

***Таблица 17.1. Аргументы метода*** *`Expr.map_batches()`*

| Аргумент | Описание |
|---|---|
| `Function` | Функция для применения к объекту Series |
| `return_dtype` | Тип данных объекта Series, возвращаемого функцией |
| `is_elementwise` | Индикатор того, может ли функция применяться к отдельным элементам. Если да, то она может быть запущена в потоковом движке, но может возвращать некорректные результаты в контексте GroupBy |
| `agg_list` | Агрегирует значения выражения в список перед применением функции в контексте GroupBy. В этом случае функция будет вызвана один раз для списка групп, а не по разу для каждой группы |

Давайте посмотрим на использование метода `Expr.map_batches()` на примере применения функции `softmax` из пакета NumPy к объектам Series `fea-`

ture1 и feature2. Эта нормализующая функция преобразует список значений в вероятности, в сумме составляющие единицу:

```
import polars.selectors as cs
from scipy.special import softmax

ml_dataset = pl.DataFrame(
    {
        "feature1": [0.3, 0.2, 0.4, 0.1, 0.2, 0.3, 0.5],
        "feature2": [32, 50, 70, 65, 0, 10, 15],
        "label": [1, 0, 1, 0, 1, 0, 0],
    }
)

ml_dataset.select(
    "label",
    cs.starts_with("feature").map_batches(
        lambda x: softmax(x.to_numpy()),
    ),
)
```

Вывод:

```
shape: (7, 3)
┌───────┬──────────┬──────────────┐
│ label ┆ feature1 ┆ feature2     │
│ ---   ┆ ---      ┆ ---          │
│ i64   ┆ f64      ┆ f64          │
╞═══════╪══════════╪══════════════╡
│ 1     ┆ 0.143782 ┆ 3.1181e-17   │
│ 0     ┆ 0.130099 ┆ 2.0474e-9    │
│ 1     ┆ 0.158904 ┆ 0.993307     │
│ 0     ┆ 0.117719 ┆ 0.006693     │
│ 1     ┆ 0.130099 ┆ 3.9488e-31   │
│ 0     ┆ 0.143782 ┆ 8.6979e-27   │
│ 0     ┆ 0.175616 ┆ 1.2909e-24   │
└───────┴──────────┴──────────────┘
```

## Применение функций к группам

Метод *Expr.map_groups()* позволяет применять функцию на Python к каждой группе в контексте GroupBy.

Скажем, у нас есть датафрейм с температурами в разных странах, при этом для Америки температуры хранятся в градусах Фаренгейта, а для других стран – в градусах Цельсия. Если для вашего анализа важен лишь фактор изменения температуры, вы можете масштабировать переменную в рамках каждой группы, чтобы сделать значения сравнимыми. Для этого можно воспользоваться классом StandardScaler из библиотеки Scikit-learn, как показано ниже:

```
from sklearn.preprocessing import StandardScaler

def scale_temperature(group):
    scaler = StandardScaler()
    scaled_values = scaler.fit_transform(group[["temperature"]].to_numpy())
    return group.with_columns(
        pl.Series(values=scaled_values.flatten(), name="scaled_feature")
    )

temperatures = pl.DataFrame(
    {
        "country": ["USA", "USA", "USA", "USA", "NL", "NL", "NL"],
        "temperature": [32, 50, 70, 65, 0, 10, 15],
    }
)

temperatures.group_by("country").map_groups(scale_temperature)
```

Вывод:

```
shape: (7, 3)
┌─────────┬─────────────┬────────────────┐
│ country ┆ temperature ┆ scaled_feature │
│ ---     ┆ ---         ┆ ---            │
│ str     ┆ i64         ┆ f64            │
╞═════════╪═════════════╪════════════════╡
│ USA     ┆ 32          ┆ -1.502872      │
│ USA     ┆ 50          ┆ -0.287066      │
│ USA     ┆ 70          ┆ 1.063831       │
│ USA     ┆ 65          ┆ 0.726107       │
│ NL      ┆ 0           ┆ -1.336306      │
│ NL      ┆ 10          ┆ 0.267261       │
│ NL      ┆ 15          ┆ 1.069045       │
└─────────┴─────────────┴────────────────┘
```

Наконец, если вам необходимо осуществить полный контроль над отдельными группами в контексте GroupBy, вы можете пройтись по ним итерациями. Это может быть полезно, если вы хотите применить к каждой группе свою пользовательскую функцию или исследовать отдельные группы. Цикл по группам возвращает кортеж, состоящий из идентификатора группы и соответствующего датафрейма:

```
temperatures = pl.DataFrame(
    {
        "country": ["USA", "USA", "USA", "USA", "NL", "NL", "NL"],
        "temperature": [32, 50, 70, 65, 0, 10, 15],
    }
)

for group, df in temperatures.group_by("country"):
    print(f"{group[0]}:\n{df}\n")
```

Вывод:

```
USA:
shape: (4, 2)
┌─────────┬─────────────┐
│ country ┆ temperature │
│ ---     ┆ ---         │
│ str     ┆ i64         │
╞═════════╪═════════════╡
│ USA     ┆ 32          │
│ USA     ┆ 50          │
│ USA     ┆ 70          │
│ USA     ┆ 65          │
└─────────┴─────────────┘
NL:
shape: (3, 2)
┌─────────┬─────────────┐
│ country ┆ temperature │
│ ---     ┆ ---         │
│ str     ┆ i64         │
╞═════════╪═════════════╡
│ NL      ┆ 0           │
│ NL      ┆ 10          │
│ NL      ┆ 15          │
└─────────┴─────────────┘
```

## Кеш – наше все

В модуле `functools` присутствует очень полезный декоратор *`@lru_cache`*, позволяющий оптимизировать выполнение ресурсоемких функций. При помощи кеширования результатов вызова функций этот декоратор позволяет значительно ускорить последующие вызовы, особенно при использовании тех же аргументов. Это бывает очень полезно при применении функции к столбцу в датафрейме, содержащему повторяющиеся значения. Декоратор `@lru_cache` кеширует результаты вызова функций. При повторном обращении к функции с такими же параметрами вычисление функции не производится, а результаты берутся из кеша.

Вы можете передать декоратору `@lru_cache` аргумент `maxsize`, ограничивающий количество сохраняемых результатов. По умолчанию этот параметр установлен в 128, но вы можете увеличить его, чтобы чаще обращаться к кешу, жертвуя при этом дополнительной памятью. При переполнении декоратор `@lru_cache` удаляет из кеша недавно записанные значения. Вы можете передать аргументу `maxsize` значение `None`, если хотите хранить все результаты, но имейте в виду, что в этом случае у вас очень быстро может закончиться место в оперативной памяти. Очистить кеш при необходимости можно при помощи функции `cache_clear()`.

Вот как можно применить декоратор `@lru_cache` в задаче анализа тональности текста, которую мы решали ранее:

```
from functools import lru_cache
from textblob import TextBlob

@lru_cache(maxsize=256)
```

```
def analyze_sentiment(review):
    return TextBlob(review).sentiment.polarity

reviews = pl.DataFrame(
    {
        "reviews": [
            "This product is great!",
            "Terrible service.",
            "Okay, but not what I expected.",
            "Excellent! I love it.",
        ]
    }
)

reviews.with_columns(
    sentiment_score=pl.col("reviews").map_elements(
        analyze_sentiment, return_dtype=pl.Float64
    )
)
```

Вывод:

```
shape: (4, 2)
┌──────────────────────────┬─────────────────┐
│ reviews                  ┆ sentiment_score │
│ ---                      ┆ ---             │
│ str                      ┆ f64             │
╞══════════════════════════╪═════════════════╡
│ This product is grea…    ┆ 1.0             │
│ Terrible service.        ┆ -1.0            │
│ Okay, but not what I…    ┆ 0.2             │
│ Excellent! I love it…    ┆ 0.75            │
└──────────────────────────┴─────────────────┘
```

Обратите внимание, что в нашем примере мы никак не улучшили производительность, поскольку в данных у нас присутствуют только уникальные значения.

## Применение функций к выражениям

Метод *Expr.pipe()* предоставляет структурированный подход к применению пользовательских функций на Python к выражениям. Первым аргументом функция на языке Python принимает выражение, к которому применяется, и возвращает она также выражение.

Вот небольшой пример:

```
addresses = pl.DataFrame(
    {
        "address": [
            "Nieuwezijds Voorburgwal 147",
            "Museumstraat 1",
            "Oosterdok 2",
```

```
        ]
    }
)

def extract_house_number(input_expr: pl.Expr) -> pl.Expr:
    """Извлекаем номер дома из строкового адреса"""
    return input_expr.str.extract(r"\d+", 0).cast(pl.Int64)

addresses.with_columns(
    house_numbers=pl.col("address").pipe(extract_house_number)
)
```

Вывод:

```
shape: (3, 2)
┌──────────────────────┬──────────────┐
│ address              ┆ house_numbers │
│ ---                  ┆ ---          │
│ str                  ┆ i64          │
╞══════════════════════╪══════════════╡
│ Nieuwezijds Voorburg… ┆ 147          │
│ Museumstraat 1       ┆ 1            │
│ Oosterdok 2          ┆ 2            │
└──────────────────────┴──────────────┘
```

## Применение функций к датафреймам и ленивым датафреймам

Метод *df.pipe()* можно использовать для применения функций на Python к целым датафреймам и ленивым датафреймам. При этом рекомендуется использовать именно ленивые датафреймы, поскольку в этом случае может применяться оптимизация. Функция в качестве первого аргумента должна принимать датафрейм, как показано ниже:

```
small_numbers = pl.DataFrame({"ints": [2, 4, 6], "floats": [10.0, 20.0, 30.0]})

def scale_the_input(
    df: pl.DataFrame | pl.LazyFrame, scale_factor: int
) -> pl.DataFrame | pl.LazyFrame:
    """Масштабируем входной датафрейм в соответствии с переданным множителем"""
    return df * scale_factor

small_numbers.pipe(scale_the_input, 5)
```

Вывод:

```
shape: (3, 2)
┌──────┬────────┐
│ ints ┆ floats │
│ ---  ┆ ---    │
│ f64  ┆ f64    │
```

```
╞══════╪═══════╡
│ 10.0 ┆ 50.0  │
│ 20.0 ┆ 100.0 │
│ 30.0 ┆ 150.0 │
└──────┴───────┘
```

Подытоживая, скажем, что в Polars предусмотрено сразу несколько методов для применения функций на языке Python к данным. Это методы `Expr.map_elements()`, `Expr.map_batches()`, `Expr.map_groups()`, `Expr.pipe()` и `df.pipe()`. Хотя пользовательские функции могут позволить вам выполнить практически любые вычисления, необходимо помнить, что в плане эффективности они практически всегда будут уступать родным выражениям Polars. Если же вам не обойтись без внешних функций, но при этом на вход они могут получать одни и те же значения, рассмотрите возможность использования декоратора `@lru_cache`. Понимание всех тонкостей применения разных инструментов, доступных в Polars, позволит вам добиться оптимальной эффективности выполняемых операций.

# Регистрация своего пространства имен

Как мы уже знаем, в Polars присутствует достаточно много встроенных пространств имен, таких как `str`, с помощью которых можно выполнять соответствующие действия (`pl.col(...).str....()` и т. д.), но это не мешает вам создавать свои собственные пространства имен. После этого вы можете регистрировать свои пользовательские функции и выражения в созданных пространствах имен. Это позволит вам создать более интуитивно понятный и дружественный API для ваших пользователей.

Для создания нового пространства имен вам необходимо создать класс в Python с декоратором того уровня, которому будет принадлежать это пространство. Доступные уровни пространств имен перечислены в табл. 17.2.

***Таблица 17.2. Декораторы для создания пространств имен***

| Контекст | Декоратор | Пример использования |
|---|---|---|
| Выражение | `@pl.api.register_expr_namespace("...")` | `pl.col(...).<пространство имен>.<функция>` |
| Датафрейм | `@pl.api.register_dataframe_namespace("...")` | `df.<пространство имен>.<функция>` |
| Ленивый датафрейм | `@pl.api.register_lazy_frame_namespace("...")` | `lf.<пространство имен>.<функция>` |
| Объект Series | `@pl.api.register_series_namespace("...")` | `col.<пространство имен>.<функция>` |

Способы регистрации пространств имен для разных уровней не отличаются. Вы создаете класс с соответствующим декоратором и определяете функции, которые хотите зарегистрировать в этом классе.

Рассмотрите следующий код, в котором мы регистрируем пользовательское пространство имен `celsius` для выражения `pl.col(...).celsius....()`:

```
@pl.api.register_expr_namespace("celsius") ❶
class Celsius:
    def __init__(self, expr: pl.Expr): ❷
        self._expr = expr

    def to_fahrenheit(self) -> pl.Expr: ❸
        return (self._expr * 9 / 5) + 32

    def to_kelvin(self) -> pl.Expr:
        return self._expr + 273.15
```

❶ В этой строке кода мы регистрируем пользовательское пространство имен уровня выражения с именем `celsius`.
❷ Конструктор класса принимает на вход выражение и сохраняет его во внутреннем атрибуте. Это позволяет использовать выражение, передаваемое в пользовательскую функцию в созданном пространстве имен.
❸ Метод `Celsius.to_fahrenheit()` принимает на вход выражение, переданное нашему пространству имен, и возвращает новое выражение, преобразующее температуру в Фаренгейты.

В своем коде вы можете использовать пользовательское пространство имен так, как показано ниже:

```
temperatures = pl.DataFrame({"celsius": [0, 10, 20, 30, 40]})

temperatures.with_columns(fahrenheit=pl.col("celsius").celsius.to_fahrenheit())
```

Вывод:

```
shape: (5, 2)
┌─────────┬────────────┐
│ celsius ┆ fahrenheit │
│ ---     ┆ ---        │
│ i64     ┆ f64        │
╞═════════╪════════════╡
│ 0       ┆ 32.0       │
│ 10      ┆ 50.0       │
│ 20      ┆ 68.0       │
│ 30      ┆ 86.0       │
│ 40      ┆ 104.0      │
└─────────┴────────────┘
```

Здесь мы создали датафрейм с одним столбцом `celsius`, после чего преобразовали значения в этом столбце в градусы по Фаренгейту с помощью пользовательского пространства имен `celsius`.

**Никаких автодополнений**

Поскольку пользовательские пространства имен не являются частью пакета Polars и создаются во время выполнения, они не распознаются средами интерактивной разработки, что не позволяет использовать применительно к ним автодополнение син-

таксиса. Это означает, что вы не будете получать никаких подсказок в ноутбуке или IDE при работе с такими пространствами.

# Плагины Polars в Rust

Если вы хотите выжать максимум возможного из библиотеки Polars, то можете писать свои плагины на Rust и использовать их в процессе разработки. Это позволит движку Polars осуществлять *динамическое связывание* ваших пользовательских функций и выражений в процессе выполнения. Динамическое связывание дает возможность Polars бесшовно интегрировать внешний код в свои пакеты без необходимости повторной компиляции основного пакета. Это позволит плагину использовать оптимизации, параллелизм и эффективные механизмы, заложенные в Rust, и выполняться столь же быстро, как и родные функции Polars, с минимальными накладными расходами. Таким образом, мы считаем именно этот способ оптимальным вариантом расширения функционала Polars.

Чтобы понять, как это все работает, давайте создадим базовую пользовательскую функцию, которая будет заменять все значения в объекте Series строкой `"Hello, world!"`.

## Подготовка

Поскольку мы будем программировать на языке Rust, нам необходимо сначала установить его. Это можно сделать, воспользовавшись инструкцией на официальном сайте по адресу *https://www.rust-lang.org/tools/install*. Для проверки корректности установки вы можете запустить следующую команду:

```
! rustc --version
```

Если вы увидите версию Rust, значит, все в порядке и можно двигаться дальше.

## Анатомия проекта с плагином

Создание пользовательского выражения предполагает наличие отдельной папки с проектом со следующими ключевыми файлами:

```
/
├── src
│   ├── expressions.rs
│   └── lib.rs
├── hello_world_func
│   └── __init__.py
├── Cargo.toml
```

Здесь мы видим такие файлы:

- в файле *src/expressions.rs* содержится код на языке Rust для пользовательского выражения;
- в файле *src/lib.rs* находится код на Rust, определяющий пользовательское выражение как пакет;
- файл *hello_world_func/init.py* содержит код на Python, регистрирующий функцию в движке Polars;
- в файле *Cargo.toml* находятся метаданные проекта Rust, такие как название, версия и зависимости.

## Плагин

Давайте посмотрим, что будет содержаться в перечисленных файлах для функции, заменяющей все значения в объекте Series строкой `"Hello, world!"`. Первый файл будет содержать код на Rust для пользовательского выражения, показанный ниже:

*src/expressions.rs*

```
use polars::prelude::*; ❶
use pyo3_polars::derive::polars_expr;

#[polars_expr(output_type=String)] ❷
fn hello_world(inputs: &[Series]) -> PolarsResult<Series> { ❸
    // Функция принимает на вход объект Series и возвращает новый Series
    // той же длины, заполненный значениями "Hello, world!"
    let length = inputs[0].len(); ❹
    let result: Vec<String> = vec!["Hello, world!".to_string(); length]; ❺
    Ok(Series::new("hello_world".into(), result)) ❻
}
```

❶ Сначала мы импортируем пакеты Polars и PyO3 Polars для Rust.
❷ Эта строка работает подобно декоратору в Python и позволяет функции на Rust регистрироваться как функции Polars, возвращающей значение с типом String. Это даст возможность Polars оптимизировать выполнение функции для этого вывода.
❸ На вход функции поступает срез объекта Series, а на выход возвращается объект типа `PolarsResult` с Series. `PolarsResult` – это объект, который может содержать либо значение, либо ошибку.
❹ Эта переменная принимает длину входящего объекта Series.
❺ Здесь создается вектор нужной длины, в котором каждый элемент представляет собой строку `"Hello, world!"`.
❻ Возвращается из функции новый объект Series с именем `hello_world`, содержащий созданный ранее вектор строк.

Теперь рассмотрим содержимое файла, определяющего пользовательское выражение как пакет:

*src/lib.rs*

```
mod expressions; ❶

use pyo3::types::PyAnyMethods; ❷
use pyo3::types::PyModule;
use pyo3::{pymodule, Bound, PyResult};

#[pymodule] ❸
fn hello_world_func(m: &Bound<'_, PyModule>) -> PyResult<()> { ❹
    m.setattr("__version__", env!("CARGO_PKG_VERSION"))?; ❺
    Ok(()) ❻
}
```

❶ В этой строке мы указываем Rust на необходимость включения файла *expressions.rs* в модуль.
❷ Импорты, позволяющие PyO3 настроить интерфейс между Python и Rust.
❸ Макрос Rust, который помечает следующую за ним функцию как функцию инициализации Python. Этот код вызывается для инициализации модуля.
❹ Функция, выступающая в роли инициализатора.
❺ Добавляет тег с версией к модулю на основе версии пакета Cargo.
❻ Оповещает об успешном выполнении функции.

Файл инициализации запускается в момент загрузки плагина:

*hello_world_func/__init__.py*

```
from pathlib import Path

import polars as pl
from polars.plugins import register_plugin_function
from polars.type_aliases import IntoExpr

PLUGIN_PATH = Path(__file__).parent

def hello_world(expr: IntoExpr) -> pl.Expr: ❶
    return register_plugin_function( ❷
        plugin_path=PLUGIN_PATH, ❸
        function_name="hello_world", ❹
        args=expr, ❺
        is_elementwise=True, ❻
    )
```

❶ Функция принимает аргумент с типом `IntoExpr`, который передает в плагин на Rust, и возвращает выражение, зарегистрированное в Polars. `IntoExpr` представляет собой тип, который является или может быть преобразован в выражение Polars. Примером того, что может быть преобразовано в выражение Polars, может служить значение типа `String`, которое может быть трансформировано в выражение `pl.col("column_name")`.
❷ Регистрируем плагин как функцию Polars.
❸ Путь, где располагается плагин Rust.
❹ Имя функции Rust для регистрации.

❺ Передаем выражение в плагин Rust. Можно также передать несколько аргументов плагину.
❻ Указывает на то,что функция является поэлементной. Это означает, что действия будут выполняться с каждым элементом объекта Series по отдельности. Это, в свою очередь, позволит Polars использовать оптимальные алгоритмы при выполнении кода.

И наконец, файл с метаданными проекта Rust:

*Cargo.toml*

```
[package]
name = "hello_world_plugin"
version = "1.0.0"
edition = "2021"

[lib]
name = "hello_world_func"
crate-type = ["cdylib"]

[dependencies]
polars = { version = "*" }
pyo3 = { version = "*", features = ["extension-module"] }
pyo3-polars = { version = "*", features = ["derive"] }
serde = { version = "*", features = ["derive"] }
```

В этом файле содержатся метаданные проекта Rust. Среди прочего мы указываем имя, версию, редакцию и зависимости, используемые в проекте. Также мы сообщаем имя и тип пакета. Поле `name` в секции `[lib]` должно соответствовать имени папки с проектом Rust. В нашем случае это `hello_world_func`.

## Компиляция плагина

Чтобы созданный плагин оказался доступен в вашем коде на Python, вам нужно скомпилировать код на Rust следующей командой:

```
! cd plugins/hello_world_plugin && uv run maturin develop --release
```

**Флаг --release**

Флаг `--release` оптимизирует компилируемый код с точки зрения его быстродействия, жертвуя при этом скоростью компиляции. При оценке производительности плагина всегда используйте этот флаг. В процессе разработки, когда вам важнее быстро компилировать проект, флаг можно опустить.

## Оценка быстродействия

Теперь, когда код скомпилирован, вы можете грубо оценить эффективность пользовательского выражения, чтобы понять, какого прироста в скорости мы добились. Это можно сделать при помощи следующего кода на Python:

```
import polars as pl
from hello_world_func import hello_world ❶
import time

lots_of_strings = pl.DataFrame(
    {
        "a": ["1", "2", "3", "4"] * 100_000,
    }
)

times = []
for i in range(10):
    t0 = time.time()
    out = lots_of_strings.with_columns(
        pl.col("a").str.replace_all(r".*", "Hello, world!")
    )
    t1 = time.time()
    times.append(t1 - t0)
print(
    f"Замена строк родными методами в Polars: {sum(times) / len(times):.5f}"
) ❷

times = []
for i in range(10):
    t0 = time.time()
    out = lots_of_strings.with_columns(hello_world("a")) ❸
    t1 = time.time()
    times.append(t1 - t0)
print(f"Наша пользовательская замена строк: {sum(times) / len(times):.5f}")
```

❶ Здесь мы импортируем пользовательскую функцию. Имя модуля используется такое же, как значение поля `name` в секции `[lib]` в метаданных проекта Rust.
❷ Мы воспользовались указателем формата для вывода пяти знаков после запятой.
❸ Здесь мы вызываем пользовательскую функцию для замены всех значений в столбце строкой `"Hello, world!"`.

Вывод:

```
Замена строк родными методами в Polars: 0.04523
Наша пользовательская замена строк: 0.01301
```

Как видите, пользовательское выражение оказалось куда быстрее встроенного выражения Polars.

## Регистрация аргументов

Теперь, когда у вас есть основное понимание того, как создавать пользовательские выражения, давайте рассмотрим разные способы их использования.

## *Работа с множественными аргументами*

Вы можете передавать плагину Rust несколько аргументов для создания функций, способных оперировать одновременно с несколькими объектами Series. Это можно реализовать путем передачи списка аргументу `args` функции `register_plugin_function()`:

```
def args_func(arg1: IntoExpr, arg2: IntoExpr) -> pl.Expr:
    return register_plugin_function(
        plugin_path=PLUGIN_PATH,
        function_name="args_func",
        args=[arg1, arg2],
    )
```

Дополнительно вы можете передавать плагину Rust ключевые аргументы, присвоив параметру `kwargs` функции `register_plugin_function()` словарь, как показано ниже:

```
def kwargs_func(
    expr: IntoExpr,
    float_arg: float,
    integer_arg: int,
    string_arg: str,
    boolean_arg: bool,
) -> pl.Expr:
    return register_plugin_function(
        plugin_path=PLUGIN_PATH,
        function_name="kwargs_func",
        args=expr,
        kwargs={
            "float_arg": float_arg,
            "integer_arg": integer_arg,
            "string_arg": string_arg,
            "boolean_arg": boolean_arg,
        },
    )
```

При этом стоит не забывать прописывать ожидаемые аргументы на стороне Rust. Для этого необходимо использовать в Rust переменную типа `struct`, содержащую ожидаемые аргументы, и добавить декоратор `derive` с признаком `Deserialize` из крейта `serde`. *Крейт* (crate) представляет собой пакет в Rust, похожий на пакет в Python.

```
// Создаем свою структуру с надлежащей схемой для передачи в пользовательское выражение
#[derive(Deserialize)]
pub struct MyKwargs {
    float_arg: f64,
    integer_arg: i64,
    string_arg: String,
    boolean_arg: bool,
}
```

```
// Если вы хотите принимать ключевые аргументы, определите второй аргумент kwargs
// Вы можете передавать любую пользовательскую структуру,
// десериализуемую с помощью протокола pickle (на стороне Rust) ❶
#[polars_expr(output_type=String)]
fn append_kwargs(input: &[Series], kwargs: MyKwargs) -> PolarsResult<Series> {
    let input = &input[0];
    let input = input.cast(&DataType::String)?;
    let ca = input.str().unwrap();

    Ok(ca
        .apply_into_string_amortized(|val, buf| {
            write!(
                buf,
                "{}-{}-{}-{}-{}",
                val,
                kwargs.float_arg,
                kwargs.integer_arg,
                kwargs.string_arg,
                kwargs.boolean_arg
            )
                .unwrap()
        })
        .into_series())
}
```

❶ *pickle* описывает двоичные протоколы для сериализации и десериализации объектов Python. С помощью него мы можем десериализовать объект Python и использовать его на стороне Rust. *Десериализация* (deserializing) означает преобразование сериализованных данных обратно в формат, пригодный для использования.

## *Дополнительные аргументы*

Вы можете передавать дополнительную информацию, позволяющую Polars оптимизировать способ запуска плагина. Примеры аргументов:

- `is_elementwise`: наиболее простой и эффективный способ использования пользовательских выражений состоит в его применении к каждому элементу объекта Series независимо. Это позволяет использовать высокую степень параллелизма и применять быстрые алгоритмы. Аргумент `is_elementwise=True` дает понять, что функция должна выполняться поэлементно;

**Правильно указывайте аргументы**

При неправильном использовании аргумента `is_elementwise` Polars не выдаст никаких ошибок, но поведение функции в этом случае может оказаться непредсказуемым. Так бывает при использовании группировки или оконных функций. Если функция выполняется не поэлементно, а помечена как поэлементная, она может проигнорировать операции группировки или вычисления окон и вернуть некорректный результат. Обратное тоже справедливо: если поэлементная функция объявлена как не поэлементная, результат может оказаться неверным.

- `returns_scalar`: если функция возвращает список с одним элементом, установка параметра `returns_scalar=True` поможет распаковать список и вернуть сам элемент. К примеру, если у нас есть функция `sum()`, суммирующая элементы в списке `[1,2,3]`, в обычном виде она вернет результат в виде списка `[6]`. Установка аргумента `returns_scalar=True` позволит вернуть скалярное значение 6;
- `cast_to_supertype`: если ваш плагин принимает множество аргументов разных типов, вы можете воспользоваться аргументом `cast_to_supertype=True` для приведения их к единому типу. Это позволит избежать ошибок, связанных с типами данных, и обеспечит корректное выполнение функции. К примеру, если функция принимает аргументы типа `Int64` и `Float64`, установка аргумента `cast_to_supertype=True` позволит привести типы `Int64` к `Float64` перед запуском функции;
- `input_wildcard_expansion`: некоторые выражения представляют множественные объекты Series (например, `pl.col(*)`). При установке параметра `input_wildcard_expansion=True` выражения будут развернуты в список объектов Series;
- `pass_name_to_apply`: если установить этот параметр в `True`, объект Series, переданный в функцию, при выполнении операции GroupBy будет содержать свой набор имен. Если ваш плагин требует имя объекта Series, вы можете передать аргумент `pass_name_to_apply=True`. По умолчанию этот аргумент отключен, поскольку он требует выделения дополнительной памяти для каждой группы.

## Использование крейта Rust

Одно из преимуществ плагинов Rust заключается в возможности использования любых крейтов Rust в ваших пользовательских выражениях. Это позволяет использовать в выражениях любые пакеты Rust, совместимые с движком Polars. Для демонстрации этих возможностей мы создадим пространство имен `geo` для определения того, входит ли заданная географическая координата в полигон, построенный на основе списка координат.

## Пример: geo

С целью закрепления всех полученных знаний на практике мы создадим собственное пространство имен `geo`, содержащее две функции: одна для определения того, входит ли точка в полигон, с помощью плагина Rust, а вторая – для вычисления расстояния на сфере между двумя точками с применением формулы гаверсинуса. Это довольно типичные задачи в геопространственном анализе, и при их решении мы увидим, как можно использовать крейт Rust в ваших пользовательских выражениях. Для начала добавим крейт `geo` в файл *Cargo.toml*. Затем напишем код на Rust с нашими географическими

вычислениями. После этого обеспечим доступ к этим вычислениям из нашего кода на Python и создадим свое пространство имен для размещения этого функционала.

## Добавление крейта geo

Для добавления крейта `geo` в проект Rust необходимо поместить его в раздел `[dependencies]` файла *Cargo.toml*:

```
[package]
name = "polars_geo"
version = "1.0.0"
edition = "2021"

[lib]
name = "polars_geo"
crate-type = ["cdylib"]

[dependencies]
geo = "*"
polars = { version = "*" }
pyo3 = { version = "*", features = ["extension-module", "abi3-py38"] }
pyo3-polars = { version = "*", features = ["derive"] }
```

Здесь мы создали новый пакет с именем `polars_geo` и добавили крейт `geo` в раздел с зависимостями. Он указывает на пакет `polars_geo` и говорит о том, что это пакет cdylib. Поскольку мы добавили крейт в зависимости, он будет загружаться и компилироваться при сборке. Теперь можно поработать с добавленным крейтом в Rust.

## Код на Rust

В нашем коде на языке Rust должны выполняться следующие действия: мы принимаем входные параметры, преобразуем их в точку и полигон, определяем, принадлежит ли заданная точка полигону, и возвращаем соответствующее булево значение. Если вы раньше не работали с языком Rust, вам приведенный ниже код может показаться пугающим. Не волнуйтесь, мы подробно разберем его. Начнем с файла *expressions.rs*:

```
use geo::{coord, Contains, Distance, Haversine, Point, Polygon};
use polars::prelude::*;
use polars::series::amortized_iter::AmortSeries;
use pyo3_polars::derive::polars_expr;

fn extract_point(point_opt: Option<AmortSeries>) -> Option<Point> {
    point_opt.and_then(|point| {
        let ca = point.as_ref().f64().ok()?;
        Some(Point::new(ca.get(0)?, ca.get(1)?))
    })
}
```

```
fn extract_polygon(polygon_opt: Option<AmortSeries>) -> Option<Polygon> {
    polygon_opt.and_then(|polygon| {
        let lst = polygon.as_ref().list().ok()?;
        let coords: Option<Vec<_>> = lst
            .amortized_iter()
            .map(|coord| {
                let coord_binding = coord?;
                let ca = coord_binding.as_ref().f64().ok()?;
                Some(coord! { x: ca.get(0)?, y: ca.get(1)? })
            })
            .collect();
        coords.map(|c| Polygon::new(c.into(), vec![]))
    })
}

fn geo_point_in_polygon(
    point_opt: Option<AmortSeries>,
    polygon_opt: Option<AmortSeries>,
) -> Option<bool> {
    let point = extract_point(point_opt);
    let polygon = extract_polygon(polygon_opt);
    match (point, polygon) {
        (Some(p), Some(poly)) => Some(poly.contains(&p)),
        _ => None, // Возвращаем None, если не смогли извлечь точку или полигон
    }
}

#[polars_expr(output_type=Boolean)]
fn point_in_polygon(inputs: &[Series]) -> PolarsResult<Series> {
    let point_series = inputs[0].list()?;
    let polygon_series = inputs[1].list()?;

    let out: BooleanChunked = point_series
        .amortized_iter()
        .zip(polygon_series.amortized_iter())
        .map(|(point_opt, polygon_opt)| match (point_opt, polygon_opt) {
            (Some(point), Some(polygon)) => {
                geo_point_in_polygon(Some(point), Some(polygon))
            }
            _ => None,
        })
        .collect();

    Ok(out.into_series())
}

fn geo_haversine_distance(
    from_opt: Option<AmortSeries>,
    to_opt: Option<AmortSeries>,
) -> Option<f64> {
    let from_point = extract_point(from_opt);
    let to_point = extract_point(to_opt);
    match (from_point, to_point) {
```

```
        (Some(from_point), Some(to_point)) => {
            Some(Haversine::distance(from_point, to_point))
        }
        _ => None, // Возвращаем None, если не смогли извлечь точку
    }
}

#[polars_expr(output_type=Float64)]
fn haversine_distance(inputs: &[Series]) -> PolarsResult<Series> {
    let from_series = inputs[0].list()?;
    let to_series = inputs[1].list()?;

    let out: Float64Chunked = from_series
        .amortized_iter()
        .zip(to_series.amortized_iter())
        .map(|(from_opt, to_opt)| match (from_opt, to_opt) {
            (Some(from_point), Some(to_point)) => {
                geo_haversine_distance(Some(from_point), Some(to_point))
            }
            _ => None,
        })
        .collect();

    Ok(out.into_series())
}
```

Здесь функция `point_in_polygon()` может быть доступна в Polars в виде плагина. В ней используются три вспомогательные функции для извлечения входных значений и вызова метода `contains()` для определения вхождения точки в полигон.

Давайте пройдем по этому коду по шагам:

```
use geo::{coord, Contains, Distance, Haversine, Point, Polygon}; ❶
use polars::prelude::*; ❷
use polars::series::amortized_iter::AmortSeries; ❸
use pyo3_polars::derive::polars_expr; ❹
```

Это импорты, или декларации `use`, как они называются в Rust.

❶ Импортируем необходимые компоненты из крейта `geo` для определения вхождения точки в полигон.
❷ Импортируем нужные нам объекты из пакета Polars, такие как `Series` и `PolarsResult`.
❸ Импортируем структуру `AmortSeries` из пакета Polars для осуществления эффективных итераций по объекту Series.
❹ Импортируем макрос `polars_expr` из пакета PyO3 Polars для определения пользовательского выражения с привязками Python.

В следующем фрагменте кода мы определяем вспомогательную функцию, извлекающую точку из объекта Series:

```
fn extract_point(point_opt: Option<AmortSeries>) -> Option<Point> { ❶
    point_opt.and_then(|point| { ❷
```

```
        let ca = point.as_ref().f64().ok()?; ❸
        Some(Point::new(ca.get(0)?, ca.get(1)?)) ❹
    })
}
```

❶ Функция принимает на вход объект Series, который может содержать точку, и возвращает структуру `Point`, определенную в крейте `geo`.
❷ Функция использует метод *`and_then()`* для распаковки `Option` и применяет замыкание к значению. *Замыкание* (closure) представляет собой функцию, способную захватывать переменные в окружении, в котором она определена. Вы можете думать о замыканиях как об анонимных лямбда-функциях в Python. *`Option`* – это тип данных, который может иметь или не иметь значение, во втором случае он определяется как `None`.
❸ Функция пытается привести объект Series к типу данных `f64` и возвращает `Option` типа `f64`. Если приведение не дает результата, возвращается `None`.
❹ Функция создает новую структуру `Point` с координатами *x* и *y* на основе объекта Series, если они могут быть извлечены.

В следующем фрагменте кода мы определяем вспомогательную функцию, извлекающую полигон из объекта Series:

```
fn extract_polygon(polygon_opt: Option<AmortSeries>) -> Option<Polygon> {
    polygon_opt.and_then(|polygon| {
        let lst = polygon.as_ref().list().ok()?; ❶
        let coords: Option<Vec<_>> = lst
            .amortized_iter() ❷
            .map(|coord| {
                let coord_binding = coord?;
                let ca = coord_binding.as_ref().f64().ok()?; ❸
                Some(coord! { x: ca.get(0)?, y: ca.get(1)? })
            })
            .collect();
        coords.map(|c| Polygon::new(c.into(), vec![])) ❹
    })
}
```

❶ Вместо преобразования элементов входящего списка в тип с плавающей запятой функция обрабатывает его как список, поскольку на вход поступает список координат (`list[list[f64]]`).
❷ Метод *`amortized_iter()`* используется для эффективного прохода по списку.
❸ Считываем координаты и преобразовываем их в объект Series типа `Float64`.
❹ После успешной обработки всех координат создается новая структура с полигоном из координат.

Теперь, когда мы создали точку и полигон, пришло время проверить, входит ли данная точка в этот полигон. Сделаем это с помощью следующей вспомогательной функции:

```
fn geo_point_in_polygon(
    point_opt: Option<AmortSeries>,
    polygon_opt: Option<AmortSeries>,
) -> Option<bool> {
```

```
    let point = extract_point(point_opt); ❶
    let polygon = extract_polygon(polygon_opt); ❷
    match (point, polygon) { ❸
        (Some(p), Some(poly)) => Some(poly.contains(&p)), ❹
        _ => None, // Возвращаем None, если не смогли извлечь точку или полигон ❺
    }
}
```

❶ Извлекаем точку из входного аргумента.
❷ Извлекаем полигон из входного аргумента.
❸ Используем выражение `match`, позволяющее сравнить значения с Series из шаблонов и выполнить действие на основе совпавшего шаблона. В данном случае мы проверяем, успешно ли были извлечены точка и полигон.
❹ Если успешно, возвращается результат проверки вхождения точки в полигон.
❺ Иначе возвращается `None`.

На заключительном этапе мы создаем пользовательское выражение, использующее нашу функцию для определения вхождения точки в полигон:

```
#[polars_expr(output_type=Boolean)] ❶
fn point_in_polygon(inputs: &[Series]) -> PolarsResult<Series> { ❷
    let point_series = inputs[0].list()?; ❸
    let polygon_series = inputs[1].list()?;

    let out: BooleanChunked = point_series ❹
        .amortized_iter() ❺
        .zip(polygon_series.amortized_iter()) ❻

        .map(|(point_opt, polygon_opt)| match (point_opt, polygon_opt) {
            (Some(point), Some(polygon)) => {
                geo_point_in_polygon(Some(point), Some(polygon)) ❼
            }
            _ => None, ❽
        })
        .collect();
    Ok(out.into_series()) ❾
}
```

❶ Макрос `polars_expr` используется для определения пользовательского выражения с привязкой к Python. Здесь мы указываем, что тип выходного выражения будет булевым.
❷ Функция принимает срез объекта Series и возвращает объект типа `PolarsResult` с Series.
❸ Пытаемся преобразовать входной аргумент в список.
❹ Создаем объект `BooleanChunked` для хранения результатов.
❺ Эффективно проходим по объекту Series.
❻ Связываем объекты с точками и полигонами, чтобы можно было применить функцию `geo_point_in_polygon()` к каждой паре.
❼ Вызываем функцию `geo_point_in_polygon()` для каждой пары точек и полигонов, в случае если они были успешно извлечены.
❽ Возвращаем `None`, если не смогли извлечь точку или полигон.
❾ Возвращаем результат с типом `Ok`, говорящий об успешности выполненной операции.

После выполнения проверки на вхождение точки в полигон мы можем переходить к задаче вычисления расстояния на сфере между двумя точками с применением формулы гаверсинуса. В крейте `geo` присутствует метрика для определения этого расстояния, так что ей и воспользуемся:

```
fn geo_haversine_distance(
    from_opt: Option<AmortSeries>,
    to_opt: Option<AmortSeries>,
) -> Option<f64> {
    let from_point = extract_point(from_opt); ❶
    let to_point = extract_point(to_opt); ❶
    match (from_point, to_point) { ❷
        (Some(from_point), Some(to_point)) => {
            Some(Haversine::distance(from_point, to_point)) ❸
        }
        _ => None, // Возвращаем None, если не смогли извлечь точку
    }
}
```

❶ Извлекаем точки в виде типа `Point<f64>` с помощью вспомогательной функции `extract_point()`.
❷ Выполняем сопоставление точек с целью перехвата случаев с ошибочным извлечением.
❸ Если обе точки были успешно извлечены, вычисляем расстояние на сфере с помощью метода `haversine_distance()` из крейта `geo` и возвращаем результат в виде `Some(f64)`.

Теперь, когда мы создали вспомогательную функцию для вычисления расстояния на сфере, напишем функцию, использующую ее в процессе обработки данных:

```
#[polars_expr(output_type=Float64)] ❶
fn haversine_distance(inputs: &[Series]) -> PolarsResult<Series> { ❷
    let from_series = inputs[0].list()?; ❸
    let to_series = inputs[1].list()?;

    let out: Float64Chunked = from_series
        .amortized_iter() ❹
        .zip(to_series.amortized_iter()) ❺
        .map(|(from_opt, to_opt)| match (from_opt, to_opt) { ❻
            (Some(from_point), Some(to_point)) => {
                geo_haversine_distance(Some(from_point), Some(to_point)) ❼
            }
            _ => None, ❽
        })
        .collect();

    Ok(out.into_series()) ❾
}
```

❶ Декоратор показывает, что эта функция представляет собой выражение Polars с выходным типом `Float64`.

❷ Определяем основную функцию, которая будет зарегистрирована как выражение Polars.
❸ Извлекаем входной объект Series и преобразовываем его в список.
❹ Эффективно проходим по объекту Series с точками.
❺ Объединяем итераторы `from_series` и `to_series` с помощью метода `zip()` для параллельной обработки элементов.
❻ Выполняем сопоставление пар с соответствующими точками.
❼ Если обе точки содержат значения (являются `Some`), вызывается функция `geo_haversine_distance()` для вычисления расстояния.
❽ Если хотя бы одна из точек содержит `None`, результатом для этого элемента тоже будет `None`.
❾ Преобразуем переменную типа `Float64Chunked` в объект Series и оборачиваем в `PolarsResult`.

Теперь этот код на Rust готов к компиляции и использованию в Python. При желании увеличить быстродействие вы можете воспользоваться командой `maturin develop --release` для задействования всех оптимизаций, жертвуя скоростью сборки. Поскольку в следующем разделе мы собираемся воспользоваться нашим плагином, сделаем сборку с оптимизациями:

```
! cd plugins/polars_geo && uv run maturin develop --release
```

## *Код на Python*

Теперь, когда код на Rust скомпилирован, можно написать скрипт на Python для регистрации пользовательского выражения Polars. Для этого необходимо разместить файл со следующим содержимым в папке *polars_geo* и назвать его *init.py*:

```
from pathlib import Path

import polars as pl
from polars.plugins import register_plugin_function
from polars.type_aliases import IntoExpr

PLUGIN_PATH = Path(__file__).parent

def point_in_polygon(point: IntoExpr, polygon: IntoExpr) -> pl.Expr:
    return register_plugin_function(
        plugin_path=PLUGIN_PATH,
        args=[point, polygon],
        function_name="point_in_polygon",
        is_elementwise=True,
    )

def haversine_distance(from_point: IntoExpr, to_point: IntoExpr) -> pl.Expr:
    return register_plugin_function(
        plugin_path=PLUGIN_PATH,
        args=[from_point, to_point],
        function_name="haversine_distance",
        is_elementwise=True,
    )
```

Как и в случае с "Hello, world" ранее, в этом файле мы регистрируем пользовательские выражения в Polars. Разница состоит в том, что они оба принимают по два аргумента с типом IntoExpr. При вызове плагина Rust мы передаем оба этих аргумента в виде списка параметру args. В файле содержится чуть больше кода, чем мы привели здесь. Он также содержит код для для регистрации в пользовательском пространстве имен. Это применимо не только к плагинам, но и к любому другому коду. Давайте посмотрим, как именно.

## *Создание пользовательского пространства имен*

Теперь, когда у вас есть готовый плагин на Rust, вы можете заключить его в пользовательское пространство имен. Так можно организовать и собрать вместе схожие методы, как это реализовано под капотом Polars (например, для методов pl.col(...).str.<метод>, предназначенных для работы со строковыми значениями). В данном случае мы зарегистрируем пространство имен geo, в котором будет находиться функция для работы с плагином на Rust. Сначала зарегистрируем пользовательское пространство имен:

```
import polars as pl
import coordinates_plugin_py as coord

@pl.api.register_expr_namespace("geo") ❶
class Geo:
    def __init__(self, input_expression: pl.Expr): ❷
        self._input_expression = input_expression

    def point_in_polygon(
        self, polygon: list[list[pl.Float64]]
    ) -> pl.Expr: ❸
        return point_in_polygon(self._input_expression, polygon)

    def haversine_distance(self, to_point: list[pl.Float64]) -> pl.Expr: ❹
        return haversine_distance(self._input_expression, to_point)
```

❶ Регистрируем класс в качестве пространства имен с ключевым словом geo.
❷ В функции __init__() сохраняется все, что выражение принимает на вход. В данном случае мы предполагаем, что функция будет вызываться для объекта Series, содержащего точку. Это означает, что выражения с точками будут поступать на вход, а выражение, в результате которого получаются полигоны, будет передаваться в следующую функцию.
❸ Эта функция в вашем пространстве имен geo вызывает ваш код на Rust для проверки вхождения точки в полигон.
❹ А эта функция обращается к коду на Rust для вычисления расстояния на сфере для двух точек.

Написав небольшой тестовый скрипт, вы можете проверить работоспособность вашего плагина. Давайте создадим датафрейм с одним случаем вхождения точки в полигон, одним – когда точка не входит в полигон, и одним некорректным случаем, который должен вернуть null:

```
points_and_polygons = pl.DataFrame(
    {
        "point": [[5.0, 5.0], [20.0, 20.0], [20.0, 20.0]],
        "polygon": [
            [[0.0, 0.0], [10.0, 0.0], [10.0, 10.0], [0.0, 10.0]],
            [
                [0.0, 0.0],
                [10.0, 0.0],
                [10.0, 10.0],
            ],
            [[0.0, None], [10.0, 0.0], [10.0, 10.0], [0.0, 10.0], [0.0, 0.0]],
        ],
    }
)
```

Осталось только вызвать функцию в вашем пространстве имен. Барабанная дробь...

```
from plugins.polars_geo import polars_geo

# Применяем функцию point_in_polygon
points_and_polygons.with_columns(
    pl.col("point").geo.point_in_polygon(pl.col("polygon")).alias("in_polygon")
)
```

Вывод:

```
shape: (3, 3)
┌──────────────┬──────────────────────┬────────────┐
│ point        ┆ polygon              ┆ in_polygon │
│ ---          ┆ ---                  ┆ ---        │
│ list[f64]    ┆ list[list[f64]]      ┆ bool       │
╞══════════════╪══════════════════════╪════════════╡
│ [5.0, 5.0]   ┆ [[0.0, 0.0], [10.0, … ┆ true       │
│ [20.0, 20.0] ┆ [[0.0, 0.0], [10.0, … ┆ false      │
│ [20.0, 20.0] ┆ [[0.0, null], [10.0,… ┆ null       │
└──────────────┴──────────────────────┴────────────┘
```

Если вы все сделали правильно, вы должны увидеть такой же вывод. Так вы можете расширять функционал библиотеки Polars без риска потери быстродействия. Это целый новый дивный мир, но результат стоит затраченных сил!

В главе 1 мы уже приводили пример вычисления расстояния между точками на сфере. Если вы хотите глубже погрузиться в мир расширения функционала библиотеки Polars, мы рекомендуем вам ознакомиться с интерактивной инструкцией по написанию плагинов от Марко Горелли (Marco Gorelli) по адресу *https://marcogorelli.github.io/polars-plugins-tutorial*.

**Работайте со строками как профессионалы**

Если вы хотите попробовать писать плагины для работы с типом данных `String`, вы можете воспользоваться вспомогательным методом *`apply_to_buffer()`*, который служит для создания повторно используемых буферов для записи строк. Это позволит вам сэкономить на выделении дополнительной памяти и значительно улучшит быстродействие.

# Заключение

В этой главе мы рассмотрели разные варианты расширения стандартного функционала библиотеки Polars.

Вы узнали, что:

- можно применять пользовательские функции, написанные на Python, к вашим данным в Polars посредством методов `Expr.map_elements()`, `Expr.map_batches()`, `Expr.map_groups()`, `Expr.pipe()` и `df.pipe()`;
- можно регистрировать собственные пространства имен Polars в Python;
- для достижения наилучшего быстродействия вы можете создавать собственные плагины на Rust и использовать их в Python;
- в собственных плагинах вы можете использовать даже крейты Rust;
- существует немало способов повышения быстродействия написанных плагинов.

В следующей главе мы погрузимся во внутреннюю архитектуру Polars и увидим, как эта библиотека работает под капотом. Это поможет вам писать эффективный код и не попадать в распространенные ловушки.

# Глава 18

# Внутреннее устройство Polars

В этой главе мы окунемся в закулисье Polars и узнаем, как работают механизмы, позволяющие этой библиотеке оставаться на переднем крае эффективности анализа и обработки данных. На протяжении книги мы не раз ссылались на эту главу, говоря, что в ней вы найдете ответы на самые сложные вопросы.

В этой главе вы узнаете:

- об устройстве внутренней архитектуры Polars;
- о некоторых технологиях, использующихся при работе с памятью;
- об оптимизациях, используемых для ускорения вычислений в Polars;
- об инструментах профилирования и тестирования кода, помогающих до предела повысить эффективность операций по обработке данных.

Все инструкции по загрузке и установке требуемых файлов вы найдете в главе 2. Мы будем исходить из предположения о том, что все рабочие файлы вы поместили в директорию *data*.

Прочитав эту главу, вы поймете, что именно лежит в основе заявленного и подтвержденного быстродействия библиотеки Polars, а также научитесь использовать ее внутренние механизмы в процессе анализа данных.

## Архитектура Polars

Основой всего является код. Код, который вы вводите в Polars, имеет в своей основе *предметно-ориентированный язык* (domain-specific language – DSL). Этот язык позволяет вам в декларативной манере определять, что необходимо сделать.

Далее код, написанный на предметно-ориентированном языке, трансформируется в так называемое *промежуточное представление кода* (intermediate representation – IR), абстрактно описывающее стоящие перед кодом задачи.

С этим представлением можно ознакомиться в текстовом виде, воспользовавшись методом ленивого датафрейма *explain()*, а в графическом виде посмотреть на него можно с помощью метода *show_graph()*. Промежуточное представление кода имеет вид *направленного ациклического графа* (directed acyclic graph – DAG), где каждый узел представляет собой вычисление, а каждое ребро – поток данных. Помимо определения вычислений, промежуточное представление кода также содержит информацию о типах и форме данных, которая помогает в работе оптимизатора. Это позволяет Polars заранее определять, может ли тот или иной фрагмент кода завершиться ошибкой, и показать соответствующее предупреждение, что значительно экономит время на написание и отладку кода.

Затем промежуточное представление кода направляется оптимизатору. Оптимизатор применяет определенные правила к этому представлению с целью упрощения вычислений. Он вполне может отказаться от выполнения избыточных действий, изменить исходный порядок действий и даже заменить операции на более оптимальные. Подробнее о работе оптимизатора мы будем говорить далее.

По завершении процесса оптимизации представление передается в *движок* (engine). Движок отвечает за то, как именно будут выполняться операции. Библиотека Polars располагает сразу несколькими движками. На настоящий момент доступны полностью пакетный движок с обработкой в памяти и потоковый движок на центральном процессоре, а также пакетный движок на графическом процессоре. Пакетный движок на центральном процессоре используется по умолчанию, если не указано иное. Потоковый движок может использоваться, если данных слишком много и они не могут поместиться в оперативной памяти. Графический процессор желательно использовать на данных, которые помещаются в видеопамять, и только для выполнения операций, которые могут быть ускорены при помощи графического процессора. Подробнее о таких операциях мы поговорим в приложении.

Движок выполняет операции и возвращает результат. По умолчанию этот результат сохраняется в оперативной памяти, но может быть записан и на диск.

На рис. 18.1 показана диаграмма со всеми слоями и связями между ними.

Такая архитектура позволяет Polars выполнять запросы предельно быстро. Еще одним ускоряющим фактором является концепция хранения данных в памяти.

# Arrow

В основе эффективности и быстродействия библиотеки Polars лежит специальная система управления памятью на основе *Apache Arrow*.

Arrow (*https://arrow.apache.org*) позиционирует себя как межъязыковую платформу разработки для аналитики в памяти. По сути, Arrow представля-

ет собой независимый от языка колоночный формат хранения в памяти для плоских и иерархических данных, оптимизированный для эффективного выполнения аналитических операций на современных центральных и графических процессорах. Преимущества Arrow очевидны, и для их извлечения не требуется установка дополнительных расширений.

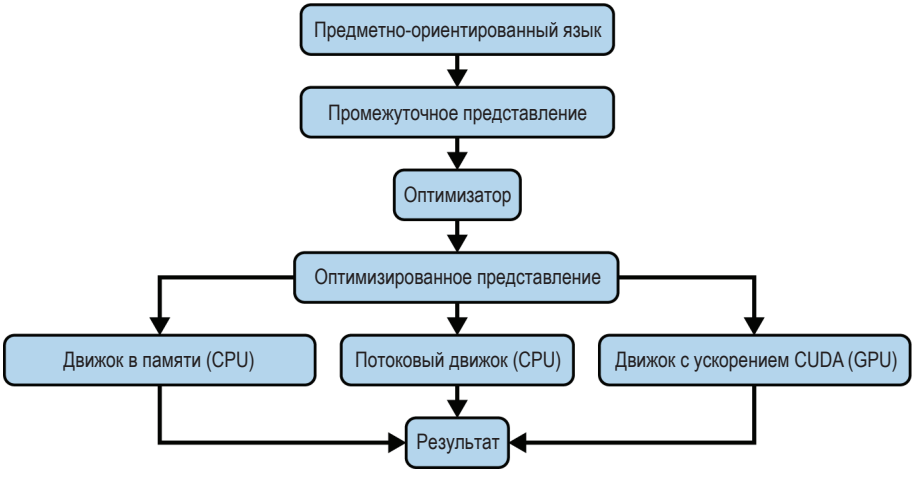



**Рис. 18.1** ❖ Концептуальная схема архитектуры Polars

Во-первых, Arrow использует колоночный формат хранения, который обеспечивает смежность расположения данных для последовательного доступа и сканирования, что позволяет читать большие объемы данных непрерывными блоками.

Кроме того, непрерывный колоночный формат отлично уживается с векторизованными операциями и позволяет использовать современную архитектуру параллельных вычислений SIMD (один поток команд и множество потоков данных (Single Instruction, Multiple Data – SIMD)), когда одна и та же операция одновременно выполняется с разными данными.

Чтобы лучше осознать эти преимущества, сравним хранение данных в Arrow с обычным шкафом для документов (см. рис. 18.2).

Представьте, что у вас есть шкаф, в котором вы храните документы о продажах товаров. В формате на основе строк в каждом ящике будут храниться все данные относительно конкретной продажи. Открыв ящик, вы сможете быстро узнать, кому был продан товар, когда, что именно за товар был продан и по какой цене. Если вам чаще всего нужно извлекать всю информацию о какой-то одной продаже, такой вариант хранения подойдет вам лучше всего.

Однако аналитические запросы чаще всего обращаются к какому-то из атрибутов всех совершенных продаж. К примеру, вам может понадобиться узнать, какие пять покупателей принесли вам наибольшую выручку. Так вы сможете как-то их поощрить. И если у вас в шкафу в каждом ящике хранится

информация об одной продаже, вам придется открыть все шкафы, чтобы посмотреть, что там за покупатель и какова сумма покупки.

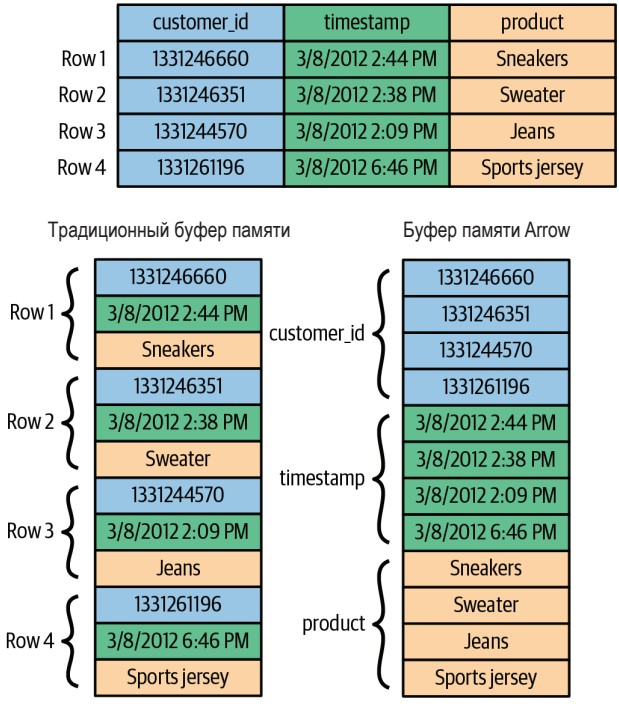



**Рис. 18.2** ❖ Буфер памяти Arrow и его преимущества в плане вычислений

Если же разместить документы в шкафу в колоночной манере, в каждом отдельном ящике будет храниться информация о каком-то конкретном атрибуте, или категории, данных. Это значит, что в одном из ящиков может быть собрана информация обо всех покупателях, а в другом – о суммах покупок.

С помощью последовательного чтения вы можете быстро перебирать папки внутри ящика с первой до последней. Это значительно ускорит процесс извлечения информации, поскольку вам не нужно будет закрывать ящик, переходить к следующему, открывать его и искать соответствующую папку. Из ящика с суммами покупок вы сможете извлечь пять самых дорогих покупок с идентификаторами покупателей, после чего, открыв ящик с контрагентами, легко вычислите имена ваших благодетелей. Таким образом, вам нужно будет открыть всего три ящика вместо многих сотен, что сэкономит вам уйму времени.

Благодаря колоночному хранению Arrow характеризуется постоянным временем доступа, или O(1), вне зависимости от объема данных. Это означает, что при любом количестве информации время, требующееся для доступа к конкретной точке данных, будет постоянным. Продолжая нашу аналогию со

шкафом для документов, это может означать, что при доступе к конкретному факту мы будем знать не только номер ящика, но и позицию папки в нем. Таким образом, нам не придется рыться в ящике в поиске нужной нам папки. Для операций, которым требуется получать доступ к определенным фактам в большом наборе данных, такой тип хранения подходит идеально.

Реализации Arrow присутствуют во многих популярных языках программирования. На момент написания книги свои реализации были в C/GLib, C++, C#, Go, Java, JavaScript, Julia, MATLAB, Python, R, Ruby и Rust. При этом сами реализации в разных языках могут существенно отличаться. К примеру, тип данных `Float16` присутствует далеко не во всех реализациях.

Во многих языках реализации Arrow позволяют использовать разделяемые изменяемые наборы данных без применения сериализации/десериализации. Обычно в разных языках данные в памяти представлены по-разному. Это значит, что для сопоставления данных в разных языках программирования они сперва десериализуются из одного формата, после чего сериализуются в другой формат. Этот процесс может занимать довольно много времени. Arrow избавляет разработчиков от такой необходимости, позволяя разным поддерживаемым языкам и реализациям общаться на одном диалекте. Распределение изменяемых наборов данных называется *межпроцессной коммуникацией* (inter-process communication – IPC). Arrow делает возможными операции чтения данных с *нулевым копированием* (zero-copy) в рамках одного процесса без необходимости выполнять сериализацию/десериализацию. Иными словами, вы можете прочитать файл в Polars и передать его другим пакетам, поддерживающим Arrow, таким как Great Tables, hvPlot или Altair, без необходимости выполнять преобразования из формата в формат.

# Многопоточные вычисления и операции SIMD

Polars изначально был оптимизирован для использования возможностей современного аппаратного обеспечения, включая многоядерные процессоры, способные работать со множеством потоков, и графические процессоры, легко справляющиеся с большим количеством однотипных базовых операций с использованием параллельных вычислений.

В то же время Python имеет ограничение в виде *глобальной блокировки интерпретатора* (Global Interpreter Lock – GIL), не позволяющей нескольким потокам выполнять код на Python одновременно[1]. Это упрощает реализацию и ускоряет работу однопоточных приложений. Однако GIL не позволяет

[1] Новые версии Python будут содержать интерпретаторы, не подверженные ограничению GIL, но это не значит, что они автоматически будут задействовать в работе все доступные ядра процессора.

программам, написанным на Python, в полной мере воспользоваться всеми преимуществами многопоточной архитектуры. Таким образом, если код на Python будет вынужден выполнять сложные множественные вычисления, он будет задействовать для этого только одно ядро процессора. В то же время библиотека Polars написана на языке Rust, что позволяет нивелировать эту проблему (помимо прочих преимуществ, например более быстрого байт-кода). Код на Polars может выполняться в параллельной манере, что позволит в полной мере задействовать все имеющиеся ресурсы.

Одним из преимуществ Polars в плане использования множества ядер является возможность применения операций *SIMD* (один поток команд и множество потоков данных (Single Instruction, Multiple Data – SIMD)). Упорядочивая, или векторизуя, данные в памяти, вы можете применять одну и ту же инструкцию к разным наблюдениям одновременно. Истоки такого вида вычислений лежат в теории обработки графики, где зачастую приходится выполнять одни и те же операции применительно к разным пикселям параллельно. Примером может являться изображение вспышки на экране, когда все пиксели одновременно прибавляют в яркости. В наши дни эта не самая новая концепция перекочевала в область обработки данных, и в Polars она очень активно используется для ускорения вычислений.

# Хранение строк в памяти

В главе 4 мы подробно говорили о типе данных `String`. Как вы помните, одной из сложностей работы с этим типом данных является переменная длина хранящихся в памяти строк. К примеру, целочисленные значения обладают фиксированной длиной, вследствие чего вам не составит труда вычислить адрес в памяти для следующего значения путем прибавления размера типа данных к текущему адресу в памяти. Для работы со строками такой подход неприменим. Длина строковых значений неизвестна заранее, так что адрес в памяти для следующей строки предсказать бывает не так просто. Все это привело к рождению новой концепции хранения строковых значений в памяти – в отличие от целочисленных значений, они хранятся в непрерывном буфере данных. Продолжительный блок в памяти, значения в котором хранятся последовательно одно за другим, называется *непрерывной областью памяти* (contiguous memory).

Макет представления хранит несколько важных атрибутов значений типа `String`:

- в байтах с порядковыми номерами от 0 до 3 хранится длина строкового значения;
- в байтах с номерами от 4 до 7 хранится копия первых четырех байт строкового значения. Это позволяет использовать различные оптимизации при поиске, поскольку в первых символах строк зачастую содержится ключевая информация для сравнения;

- в байтах с номерами от 8 до 11 хранится индекс буфера данных, в котором располагается строковое значение;
- в байтах с номерами от 12 до 15 хранится смещение относительно начала буфера данных, указывающее на начало искомого строкового значения.

Эта информация является необходимой и достаточной для извлечения строки из буфера данных без необходимости осуществлять поиск в памяти.

Также в Polars предусмотрен иной вид оптимизации для хранения строковых значений длиной менее 12 байт. Такие значения хранятся непосредственно в макете представления, а не в буфере данных. Подобный тип хранения именуется *встраиванием* (inlining). Если длина строки не превышает 12 байт, ее можно хранить с использованием всего 12 байт. Это позволяет Polars не выделять дополнительную память и не искать нужное значение в буфере данных, что может занимать немало времени.

На рис. 18.3 показана схема работы такого типа хранения. Строковое значение *Aerosmith* может быть сохранено с использованием встраивания, вследствие чего мы не задействуем буфер данных. В то же время более длинная строка *Toots and the Maytals* хранится в буфере данных начиная с 22-й позиции после строки *The Velvet UnderGround*, которая, в свою очередь, стартует с самого начала строки.

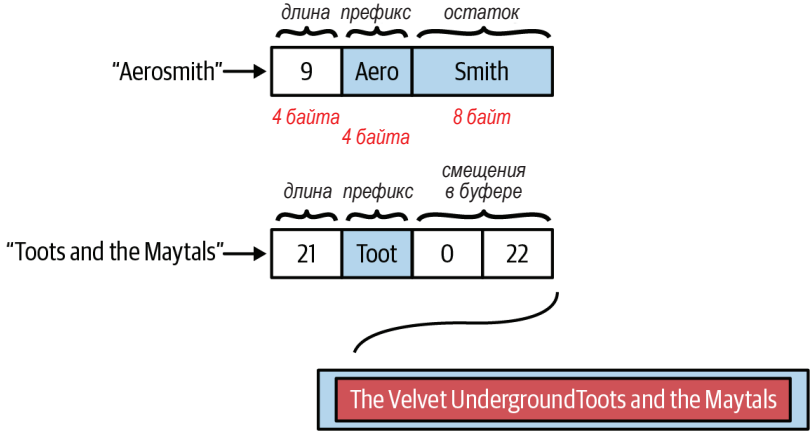



**Рис. 18.3** ❖ Хранение длинных и коротких строк в памяти

# ChunkedArrays и Series

В главе 4 мы говорили о структуре данных Series. Но это не самый низкий уровень хранения данных в Polars. Каждый объект Series (а значит, и каждый столбец в датафрейме) внутренне представлен в Polars в виде *ChunkedArray*.

ChunkedArray представляет собой контейнерный класс для хранения последовательности массивов данных. Использование ChunkedArray вместо обычных массивов, хранящих все данные, позволяет воспользоваться несколькими видами оптимизации, включая оптимизированное управление памятью. При добавлении данных в ChunkedArray данные фактически добавляются в существующий объект. Это позволяет не копировать данные в другое место и не удалять предыдущий массив, что дает немалый выигрыш во времени. Кроме того, Polars позволяет разбивать данные на блоки, с которыми можно работать по отдельности или параллельно с целью повышения быстродействия. Каждый блок данных может обрабатываться на отдельном ядре процессора, что позволяет существенно ускорить выполнение операций.

**Блоки данных**

Управление блоками данных позволяет оптимизировать работу с данными в Polars. Процесс *переразбиения* (rechunking) данных подразумевает изменение размера блока в ChunkedArray. В Polars переразбиение обычно выполняется путем помещения всех данных в один блок. В рамках каждого блока данные располагаются в памяти последовательно. В жадном режиме данные автоматически переразбиваются после чтения. Polars предполагает, что при использовании жадного режима пользователь в большинстве случаев будет анализировать полученные данные. При этом зачастую происходит множество запросов к одному и тому же датафрейму, что оправдывает затрачиваемое время на переразбиение. При использовании ленивых вычислений оптимизатор сам решает, когда лучше выполнять переразбиение.

Обычно не стоит принимать этот фактор во внимание, за исключением ситуаций, когда вы выполняете контрольную проверку быстродействия. В этом случае стоит помнить, что при передаче аргументу `rechunk` значения `True` будет выполняться не одна операция, а две.

# Оптимизация запросов

В главе 5 мы говорили о том, как использование ленивой парадигмы позволяет оптимизировать запросы, но не вдавались в подробности этого процесса.

В табл. 18.1 приведена большая часть оптимизаций, используемых в Polars на момент написания книги.

***Таблица 18.1. Некоторые оптимизации, применяемые в Polars***

| Оптимизация | Описание |
|---|---|
| Раннее выполнение предикатов | Применение фильтров на ранней стадии запроса – еще на этапе сканирования |
| Раннее выполнение проекций | Отбор только нужных столбцов на ранней стадии запроса – также на этапе сканирования |
| Раннее осуществление срезов | Загрузка только требуемых срезов данных на этапе сканирования. Не материализуйте срезы (например, командой `lf.head(10)`) |

***Таблица 18.1*** *(окончание)*

| Оптимизация | Описание |
|---|---|
| Исключение общих подпланов | Кеширование поддеревьев или результатов сканирования файлов, используемых разными поддеревьями в плане запроса |
| Упрощение выражений | Применение различных оптимизаций, таких как замена дорогостоящих операций более быстрыми альтернативами |
| Порядок объединений | Оценка ветвей при объединении таблиц на предмет определения оптимального порядка объединения с минимальным расходованием памяти |
| Приведение типов | Типы данных приводятся исходя из возможности минимизировать расход памяти |
| Оценка кардинальности | Производится оценка кардинальности таблиц для выявления оптимальной стратегии группировки данных |

Давайте рассмотрим некоторые из приведенных видов оптимизации более подробно.

## Оптимизации уровня сканирования ленивого датафрейма

Первая и очень важная группа видов оптимизации связана с загрузкой данных на этапе сканирования. *Этап сканирования* (scan level) – это слой выполнения, на котором Polars читает данные из источника. Оптимизации этого вида связаны с попытками исключить физическое чтение данных, которые нам фактически не понадобятся.

Раннее выполнение проекций означает оптимизацию запросов путем отбора нужных нам столбцов на самой ранней стадии, насколько это возможно. Это предотвращает чтение избыточных столбцов в память. В этом примере мы воспользуемся набором данных, посвященным нью-йоркским такси, с которым работали ранее в этой книге.

Мы снова попытаемся выявить трех лидеров среди компаний поставщиков услуг по отношению выручки к преодоленной дистанции, но на этот раз воспользуемся ленивым API:

```
taxis = pl.scan_parquet("data/taxi/yellow_tripdata_*.parquet") ❶
taxis.select(pl.col("trip_distance")).show_graph() ❷
```

❶ Функция *`pl.scan_parquet()`* не читает данные из файла на диске немедленно. Вместо этого она возвращает ленивый датафрейм с необходимыми метаданными, такими как схема и количество строк и столбцов. Ленивые датафреймы являются настоящим оплотом ленивого API в Polars. Методы у них почти все такие же, как у традиционных датафреймов, – разница лишь в том, что выполняются они только после вызова метода `collect()`.

❷ Здесь мы выбираем единственный столбец `trip_distance`, после чего выводим план запроса с помощью метода *`show_graph()`*, чтобы вы могли видеть, что происходит

в движке запросов. Как видно на рис. 18.4 после символа $\pi$, только одна из 19 колонок будет фактически загружена в память.

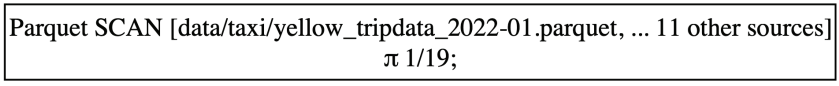



**Рис. 18.4** ❖ План выполнения запроса при сканировании набора данных и выборе единственного столбца

На рис. 18.4 показан *план выполнения запроса* (query plan). План представляет собой древовидную структуру с шагами, которые будут выполнены для достижения результата. Здесь требуется небольшое пояснение:

- первый шаг, который будет выполнен, выводится в плане последним, так что план запроса необходимо читать снизу вверх;
- каждый прямоугольник соответствует этапу плана выполнения запроса. В данном случае у нас всего один шаг, заключающийся в чтении Parquet-файла. В следующих примерах мы покажем запросы с большим количеством шагов;
- символ $\sigma$ соответствует на плане операции отбора строк и показывается при любой фильтрации;
- символ $\pi$ соответствует операции проекции столбцов и отображается при выборе определенного количества колонок.

На рис. 18.4 мы видели запись $\pi$ 1/19, что говорит о выборе одного из 19 возможных столбцов в исходном наборе данных. В плане запроса, представленном на рис. 18.5, мы видим символ $\sigma$, следом за которым идет фильтр по строкам, ограничивающий данные только теми строками, в которых значение в столбце `trip_distance` превышает 10.

Переходя к следующему примеру, поясним, что раннее выполнение предикатов – это то же самое, что и раннее выполнение проекций, но вместо отбора нужных столбцов в этом случае производится фильтрация строк. Это позволяет избежать необходимости читать избыточные данные:

```
taxis.filter(pl.col("trip_distance") > 10).show_graph()
```

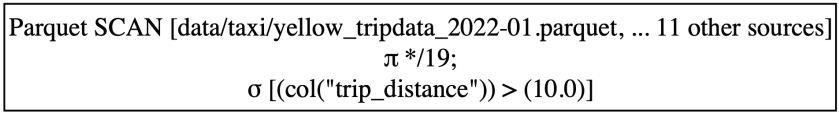



**Рис. 18.5** ❖ План выполнения запроса при сканировании набора данных и фильтрации строк

Здесь мы фильтруем набор данных по значениям в столбце `trip_distance`. На рис. 18.5 этот фильтр по строкам показан после символа $\sigma$. Указанный фильтр будет применен построчно.

Последний вид оптимизации, который мы здесь рассмотрим, связан с ранним осуществлением срезов и позволяет загружать только нужный срез после этапа сканирования (когда данные читаются в память). Подобно раннему выполнению предикатов, раннее осуществление срезов предотвращает чтение избыточных строк, но вместо чтения строк на основе фильтра строки читаются исходя из их принадлежности определенному блоку данных, как показано ниже:

```
taxis.head(2).collect()
```

Вывод:

```
shape: (2, 19)
```

| VendorID | tpep_picku p_datetime | tpep_dropo ff_datetim … | … | total_amou nt | congestion _surcharge | airport_f ee |
|---|---|---|---|---|---|---|
| i64 | --- datetime[n s] | --- datetime[n s] | | --- f64 | --- f64 | --- f64 |
| 1 | 2022-01-01 00:35:40 | 2022-01-01 00:53:29 | … | 21.95 | 2.5 | 0.0 |
| 1 | 2022-01-01 00:33:43 | 2022-01-01 00:42:07 | … | 13.3 | 0.0 | 0.0 |

В результате выполнения этой операции только две строки будут возвращены на этапе сканирования в виде датафрейма.

Показанные здесь ранние выполнения полностью исключают необходимость преобразовывать данные на поздних стадиях, если в этом нет нужды.

## Другие виды оптимизации

Оставшиеся виды оптимизации в основном связаны с эффективностью вычислений. Одним из таких видов является исключение общих подпланов. Подплан, или поддерево, представляет собой группу шагов в плане выполнения запроса. Когда определенные операции или действия по сканированию файлов используются несколькими поддеревьями в плане запроса, Polars кеширует результаты для повторного использования. Давайте начнем с того же ленивого датафрейма. Без оптимизации одни и те же наборы данных вычислялись бы повторно для разных деревьев (см. рис. 18.6):

```
values = pl.LazyFrame({"value": [10, 20, 30, 40, 50, 60]})

common_subplan = values.with_columns(pl.col("value") * 2)

branch1 = common_subplan.select(value2=pl.col("value") * 4)
```

```
branch2 = common_subplan.select(value3=pl.col("value") * 2)

combined = pl.concat([branch1, branch2])

combined.show_graph(optimized=False)
```

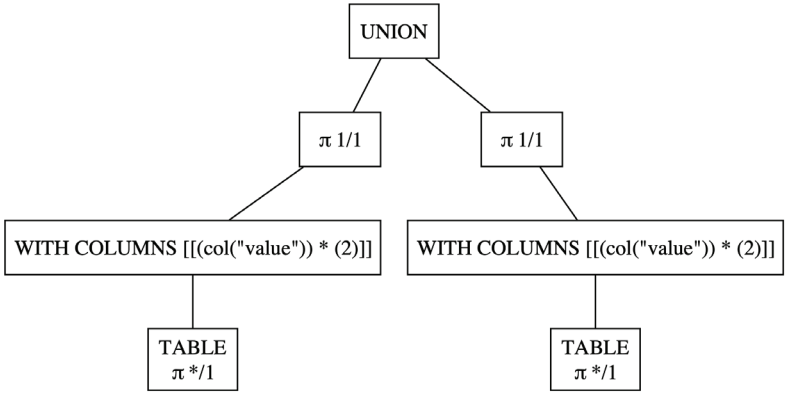



**Рис. 18.6** ❖ Неоптимизированный план выполнения запроса

С включенной оптимизацией Polars рассчитывает общие наборы данных и кеширует их для использования в разных операциях, как показано на рис. 18.7:

```
combined.show_graph()
```

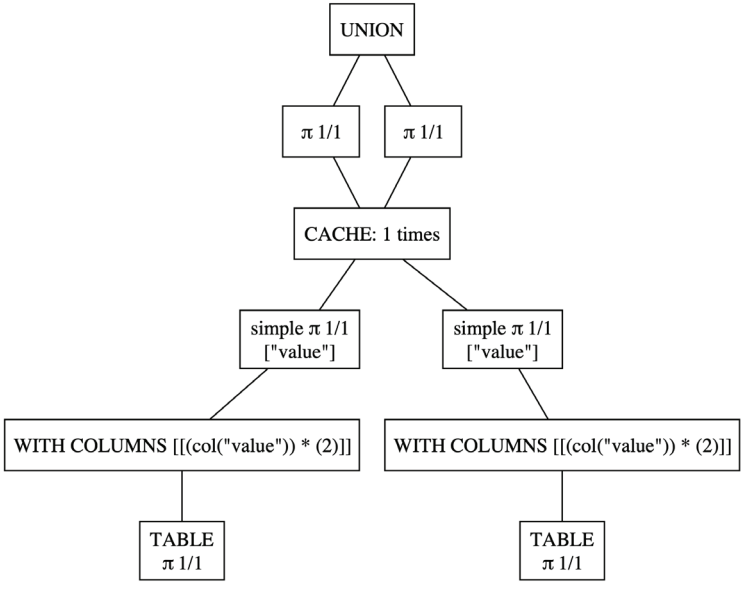



**Рис. 18.7** ❖ Оптимизированный план выполнения запроса

Это позволяет уменьшить количество чтений с диска и предотвратить двойную работу.

**Тайный ленивый режим**

Зачастую жадный API под капотом обращается к ленивому API и тут же собирает результаты. Это позволяет движку использовать некоторые оптимизации на уровне самого запроса. Кроме того, это облегчает поддержку кода, поскольку метод жадного API в этом случае является оберткой для ленивого API, что помогает избежать дублирования кода.

Еще одним приемом оптимизации является кластеризация методов `with_columns()`. Когда вы подряд используете несколько вызовов `with_columns()`, движок Polars сам пытается сократить количество вызовов, удаляя те, которые не используются. Каждый отдельный вызов метода `with_columns()` должен производиться последовательно, что может стать узким местом производительности, если их можно было бы выполнить параллельно.

Давайте создадим небольшой датафрейм с данными об индексе массы тела (BMI):

```
bmi = pl.LazyFrame(
    {"weight_kg": [70, 80, 60, 90], "length_cm": [175, 180, 160, 190]}
)
```

Теперь внесем некоторые изменения, как показано ниже:

```
bmi = (
    bmi.with_columns(weight_per_cm=pl.col("weight_kg") / pl.col("length_cm"))
    .with_columns(weight_kg_average=pl.lit(0))
    .with_columns(length_m=pl.col("length_cm") / 100)
    .with_columns(weight_kg_average=pl.col("weight_kg").mean())
)
```

В результате оптимизации запрос был приведен к следующему виду:

```
bmi = bmi.with_columns(
    weight_per_cm=pl.col("weight_kg") / pl.col("length_cm"),
    weight_kg_average=pl.col("weight_kg").mean(),
    length_m=pl.col("length_cm") / 100,
)
```

Обратите внимание, что столбец `weight_kg_average` теперь не добавляется дважды, а вызовы метода `with_columns()` были объединены вместе. Так Polars помогает оптимизировать исходный код, изначально написанный неоптимально.

# Проверка выражений

Для вывода своих выражений на новый уровень вы можете воспользоваться исключительно полезным пространством имен с названием *`Expr.meta`*. Его

можно использовать в выражениях для анализа и расширения возможностей выражений в Polars. Начнем с обзора этого пространства имен.

## Обзор пространства имен Expr.meta

В табл. 18.2 перечислены методы, доступные в пространстве имен `Expr.meta`, вместе с их описанием.

***Таблица 18.2. Методы пространства имен*** *`Expr.meta`*

| Метод | Описание |
|---|---|
| *`Expr.meta.eq()`* | Показывает, является ли выражение эквивалентным другому выражению |
| *`Expr.meta.has_multiple_outputs()`* | Показывает, расширяется ли выражение в множество выражений |
| *`Expr.meta.is_column()`* | Показывает, является ли выражение базовым (без регулярных выражений) столбцом без алиаса |
| *`Expr.meta.is_column_selection()`* | Показывает, представляет ли выражение выбор столбцов (опционально с алиасами) |
| *`Expr.meta.is_literal()`* | Показывает, является ли выражение литеральным значением |
| *`Expr.meta.is_regex_projection()`* | Показывает, расширяется ли выражение в столбцы, удовлетворяющие шаблону регулярного выражения |
| *`Expr.meta.ne()`* | Показывает, является ли выражение неэквивалентным другому выражению |
| *`Expr.meta.output_name()`* | Возвращает имя столбца, производимого этим выражением |
| *`Expr.meta.pop()`* | Извлекает последнее выражение и возвращает его входные параметры |
| *`Expr.meta.root_names()`* | Возвращает исходные имена столбцов выражения |
| *`Expr.meta.serialize()`* | Сериализует выражение в файл или строку в формате JSON |
| *`Expr.meta.show_graph()`* | Форматирует выражение в виде графика в Graphviz |
| *`Expr.meta.tree_format()`* | Форматирует выражение в виде дерева |
| *`Expr.meta.undo_aliases()`* | Отменяет операции переименования, такие как `alias` или `name.keep` |
| *`Expr.meta.write_json()`* | Записывает выражение в виде JSON |

## Примеры использования пространства имен Expr.meta

Давайте применим на практике наиболее полезные и распространенные методы пространства имен `Expr.meta`.

С помощью метода `Expr.meta.is_column()` можно быстро узнать, представляет ли выражение базовый столбец без алиаса. Это бывает особенно по-

лезно, когда вам нужно отфильтровать или проверить выражения в рамках объемного вычисления:

```
expr1 = pl.col("name")
expr2 = pl.lit("constant")
```

Вывод:

```
print(f"Выражение {expr1} - это столбец: {expr1.meta.is_column()}")
print(f"Выражение {expr2} - это столбец: {expr2.meta.is_column()}")
```

```
Выражение col("name") - это столбец: True
Выражение String(constant) - это столбец: False
```

Методом `Expr.meta.is_literal()` можно воспользоваться, чтобы узнать, является ли выражение литеральным значением. Это может помочь отличить динамические ссылки на столбцы от статических значений в выражениях:

```
print(f"Выражение {expr1} является литеральным значением: {expr1.meta.is_literal()}")
print(f"Выражение {expr2} является литеральным значением: {expr2.meta.is_literal()}")
```

Вывод:

```
Выражение col("name") является литеральным значением: False
Выражение String(constant) является литеральным значением: True
```

Метод `Expr.meta.output_name()` служит для извлечения имени производимого выражением столбца. Он бывает полезен при динамическом управлении именами столбцов, особенно после применения преобразований или алиасов:

```
expr1 = pl.col("age") * 2
expr2 = pl.col("name").alias("username")

# Получаем имена
print(f"Имя выражения {expr1}: {expr1.meta.output_name()}")
print(f"Имя выражения {expr2}: {expr2.meta.output_name()}")
```

Вывод:

```
Имя выражения [(col("age")) * (dyn int: 2)]: age
Имя выражения col("name").alias("username"): username
```

Вы можете визуализировать структуру выражения при помощи пакета Graphviz. Метод `Expr.meta.show_graph()` генерирует графическое представление, которое может пригодиться для понимания сути сложных деревьев выражений:

```
expr = (pl.col("age") * 2).alias("double_age")

expr.meta.show_graph()
```

В результате выполнения этого кода мы получим диаграмму, показанную на рис. 18.8.

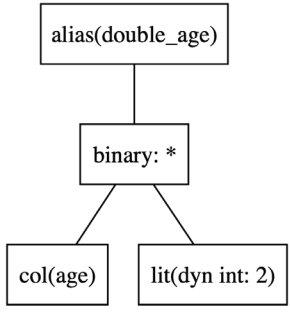



**Рис. 18.8** ❖ Графическое представление дерева операций, порождаемого выражением

Если вы использовали алиасы столбцов, а теперь хотите вернуться к исходным именам, вы можете воспользоваться методом `Expr.meta.undo_aliases()`. Это бывает полезно при подготовке выражений для операций, требующих исходных имен столбцов. Здесь мы снова можем применить метод `Expr.meta.output_name()`:

```
expr = pl.col("original_name").alias("new_name")

original_expr = expr.meta.undo_aliases()

original_expr.meta.output_name()
```

Вывод:

```
'original_name'
```

Последним мы покажем применение метода `Expr.meta.root_names()`, позволяющего извлечь исходные имена столбцов выражения:

```
expr = pl.col("origin").alias("destination")

expr.meta.root_names()
```

Вывод:

```
['origin']
```

Как видите, методы из пространства имен `Expr.meta` в Polars позволяют лучше контролировать поведение выражений и использовать в разработке принципы метапрограммирования.

# Профилирование Polars

Ленивый API в Polars содержит встроенные средства *профилирования* (profiling). При оптимизации запросов с использованием жадной парадигмы вы можете засекать время выполнения каждого шага, но в ленивой парадигме вы лишены такой возможности. Причина в том, что при использовании ленивого API запросы не выполняются до самого вызова метода `collect()`, после чего все шаги выполняются разом. Но есть выход, заключающийся в использовании метода ленивого датафрейма *`profile()`*, позволяющего сравнить быстродействие операций, входящих в запрос.

Давайте посмотрим, как этот метод работает на практике, и применим несколько преобразований к большому датафрейму, в результате которых образуется план выполнения запроса, состоящий из нескольких узлов. Воспользуемся нашим любимым датафреймом с поездками на нью-йоркских такси:

```
long_distance_taxis_per_vendor_sorted = (
    pl.scan_parquet("data/taxi/yellow_tripdata_*.parquet")
    .filter(pl.col("trip_distance") > 10)
    .select(pl.col("VendorID"), pl.col("trip_distance"), pl.col("total_amount"))
    .group_by("VendorID")
    .agg(
        total_distance=pl.col("trip_distance").sum(),
        total_amount=pl.col("total_amount").sum(),
    )
    .sort("total_distance", descending=True)
)

long_distance_taxis_per_vendor_sorted.show_graph()
```

Теперь, когда у нас есть план выполнения запроса, показанный на рис. 18.9, мы можем выполнить его профилирование следующим образом:

```
result, profiling_info = long_distance_taxis_per_vendor_sorted.profile()
```

Метод `profile()` возвращает кортеж, в котором первым располагается результирующий датафрейм, как при вызове метода `collect()`:

```
result
```

Вывод:

```
shape: (4, 3)
┌──────────┬────────────────┬──────────────┐
│ VendorID ┆ total_distance ┆ total_amount │
│ ---      ┆ ---            ┆ ---          │
│ i64      ┆ f64            ┆ f64          │
╞══════════╪════════════════╪══════════════╡
│ 2        ┆ 1.3752e8       ┆ 1.6152e8     │
```

```
| 1         | 1.3095e7        | 5.2274e7      |
| 6         | 347579.41       | 1.3881e6      |
| 5         | 1956.44         | 8702.21       |
└───────────┴─────────────────┴───────────────┘
```

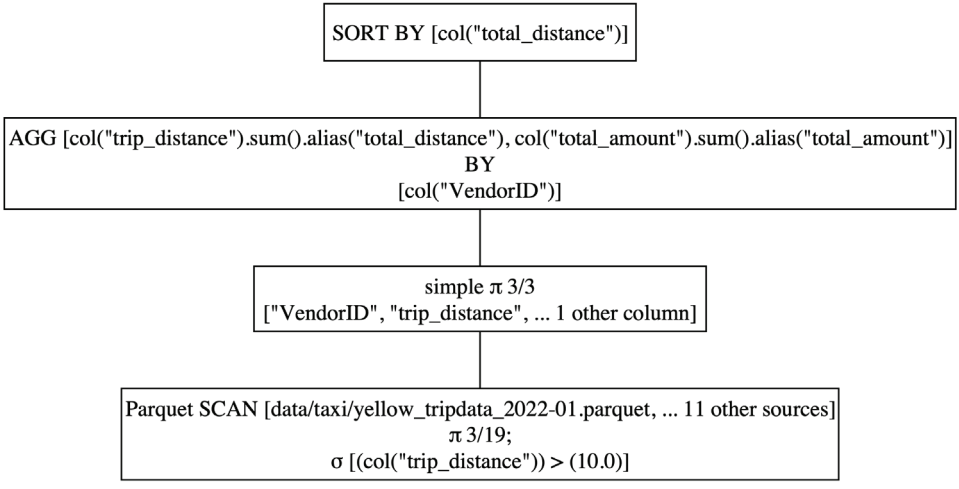



**Рис. 18.9** ❖ План выполнения запроса к набору данных по поездкам на такси

Вторым элементом в кортеже представлен еще один датафрейм – с профилирующей информацией:

```
profiling_info
```

Вывод:

```
shape: (5, 3)
┌─────────────────────────────────────────────────────┬────────┬────────┐
| node                                                | start  | end    |
| ---                                                 | ---    | ---    |
| str                                                 | u64    | u64    |
╞═════════════════════════════════════════════════════╪════════╪════════╡
| optimization                                        | 0      | 7      |
| parquet(data/taxi/yellow_tripdata_2022-01.parquet, predic… | 7      | 325779 |
| simple-projection(VendorID, trip_distance, total_amount) | 325787 | 325793 |
| group_by_partitioned(VendorID)                      | 325796 | 337054 |
| sort(total_distance)                                | 337059 | 337157 |
└─────────────────────────────────────────────────────┴────────┴────────┘
```

В этом датафрейме содержатся метки начала и окончания выполнения каждого узла в плане запроса в микросекундах. Это позволяет увидеть, какие именно узлы выполняются дольше остальных, а какие стопорят процесс. В данном случае мы видим, что наибольшим временем выполнения характеризуется узел `parquet()`, что вполне ожидаемо, поскольку на этом этапе Polars читает данные с диска.

Для более легкого восприятия вы можете воспользоваться встроенным функционалом для превращения табличных данных в графический вид посредством передачи методу `profile()` аргумента `show_plot` со значением `True`, как показано ниже:

```
long_distance_taxis_per_vendor_sorted.profile(show_plot=True, figsize=(15, 5))
```

На рис. 18.10 показана диаграмма профилирования, в которой узлы плана выполнения запроса располагаются на оси *x*, а время, требующееся на их выполнение, – на оси *y*. Вы можете изменить размер графика с помощью аргумента `figsize`, передав ему в качестве значения кортеж с шириной и высотой диаграммы в дюймах.

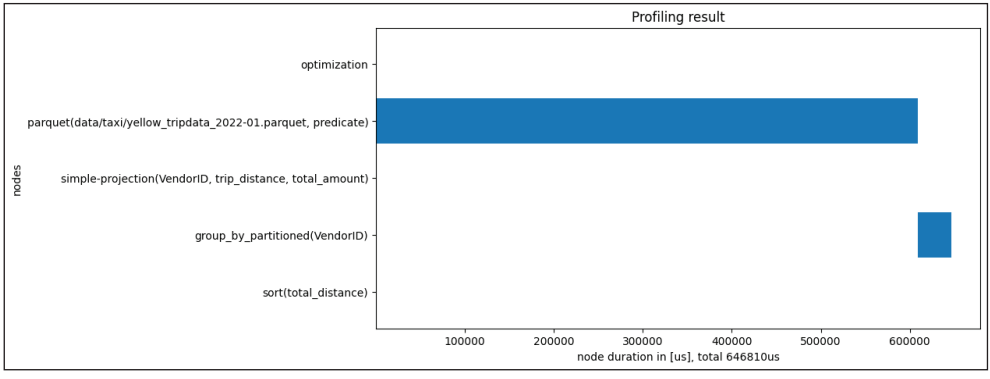



**Рис. 18.10** ❖ Диаграмма профилирования для датафрейма с поездками на нью-йоркских такси

# Тестирование в Polars

В библиотеке Polars присутствуют встроенные функции для выполнения тестирования вашего кода. Их не все сразу находят, поскольку они не импортируются с библиотекой по умолчанию, чтобы не оказывать влияния на быстродействие.

## Сравнение датафреймов и объектов Series

Для выполнения операций тестирования вы можете импортировать, к примеру, следующие четыре функции из особого модуля:

```
from polars.testing import (
    assert_series_equal,
    assert_frame_equal,
```

```
    assert_series_not_equal,
    assert_frame_not_equal,
)
```

Эти функции позволяют сравнивать два датафрейма или объекта Series и возвращать ошибку в случае их эквивалентности (или наоборот, в зависимости от предназначения функции).

Давайте создадим два датафрейма с плавающими значениями, которые затем сравним друг с другом:

```
floats = pl.DataFrame({"a": [1.0, 2.0, 3.0, 4.0]})

different_floats = pl.DataFrame({"a": [1.001, 2.0, 3.0, 4.0]})
```

Теперь сравним их:

```
assert_frame_equal(floats, different_floats)
```

Вывод:

```
AssertionError: DataFrames are different (value mismatch for column 'a')
[left]:  [1.0, 2.0, 3.0, 4.0]
[right]: [1.001, 2.0, 3.0, 4.0]
```

Как видим, было ожидаемо найдено различие в первой строке датафреймов. Но точное совпадение – это не всегда то, что вам нужно. Иногда требуется производить проверку на схожесть объектов в каком-то приближении. Для этого вы можете воспользоваться аргументами показанных выше методов, которые перечислены в табл. 18.3.

***Таблица 18.3. Аргументы функций тестирования***

| Аргумент | Описание |
|---|---|
| `check_row_order` | Требует точного соответствия порядка следования строк |
| `check_column_order` | Требует точного соответствия порядка следования столбцов |
| `check_dtypes` | Требует точного соответствия типов данных |
| `check_exact` | Требует точного совпадения значений с плавающей запятой. Если установлен в `False`, значения будут сравниваться с учетом заданных порогов (см. аргументы `rtol` и `atol`). Такое поведение распространяется только на столбцы с плавающей запятой |
| `Rtol` | Относительная погрешность для неточного сравнения. Применяется только в случае передачи аргументу `check_exact` значения `False` |
| `Atol` | Абсолютная погрешность для неточного сравнения. Применяется только в случае передачи аргументу `check_exact` значения `False` |
| `categorical_as_str` | Приводит категориальные столбцы в строковый формат перед выполнением операции сравнения. Позволяет сравнивать столбцы, не использующие один и тот же кеш со строковыми значениями, поскольку физическое кодирование столбцов может отличаться |

Если аргументы, начинающиеся с `check_*`, говорят сами за себя, то параметры с погрешностями требуют дополнительных пояснений. Эти аргументы с именами `rtol` и `atol` используются для определения того, какой может быть разница между двумя числами с плавающей запятой, чтобы их можно было считать равными.

С помощью аргумента `rtol` задается относительная погрешность, которая принимает во внимание величину значений при их сравнении и представляет собой максимально допустимое расхождение между двумя значениями, выраженное в виде доли от второго значения. На бумаге это выглядит так:

```
abs(a-b) ≤ rtol * abs(b)
```

Аргумент `atol`, напротив, задает абсолютную погрешность в виде фиксированного числового значения максимального расхождения между сравниваемыми значениями. В виде формулы это можно представить так:

```
abs(a-b) ≤ atol
```

Посмотрим, как можно использовать аргумент `rtol` на практике применительно к двум созданным выше датафреймам. Зададим относительную погрешность на уровне 0.01:

```
assert_frame_equal(floats, different_floats, rtol=0.01)
print("Датафреймы эквивалентны.")
```

Вывод:

```
Датафреймы эквивалентны.
```

В этом случае ошибка не возникла, поскольку сравниваемые значения отличаются не более чем на переданную относительную погрешность. Это бывает особенно полезно при сравнении чисел с плавающей запятой, где ошибки округления могут играть важную роль.

### Удобное представление с помощью функции `pl.from_repr()`

Чтобы сделать инициализацию датафрейма в модульных тестах более читаемой, вы можете воспользоваться удобной функцией *`pl.from_repr()`*, название которой можно расшифровать как «из (строкового) представления» (from (String) representation). Эта функция принимает строковое представление датафрейма и превращает его, собственно, в сам датафрейм. Это позволяет сохранить ожидаемый датафрейм в вашем коде в читаемом виде, что облегчает процесс тестирования.

Вы можете вставить строковое представление датафрейма (можно без строки `shape(x, y)`), а функция `pl.from_repr()` преобразует его в настоящий датафрейм, как показано ниже:

```
result = pl.DataFrame({"a": [1, 3], "b": [2, 4]}).cast(
    pl.Schema({"a": pl.Int8, "b": pl.Int8})
)
```

```
expected = pl.from_repr(
    """
┌─────┬─────┐
│ a   | b   |
| --- | --- |
| i8  | i8  |
╞═════╪═════╡
| 1   | 2   |
| 3   | 4   |
└─────┴─────┘
    """
)
assert_frame_equal(result, expected)
print("Датафреймы эквивалентны.")
```

Вывод:

```
Датафреймы эквивалентны.
```

Для более объемных датафреймов такой подход использовать не рекомендуется, поскольку это негативно скажется на читаемости вашего кода. Но на небольших датафреймах это вполне приемлемо.

# Распространенные антипаттерны

Библиотека Polars спроектирована так, чтобы самостоятельно управлять тем, *как* будут выполняться запросы, чтобы вы могли сконцентрироваться на том, *что* именно должно быть сделано. Это означает, что движок Polars сам пытается построить оптимальный план запроса, не нагружая вас лишней информацией, связанной с управлением памятью, параллелизмом и т. д. Но некоторые лазейки, позволяющие разработчику замедлить выполнение кода, все же остаются. В этом разделе мы пройдемся по некоторым антипаттернам, которых вы должны всеми силами избегать при работе с Polars.

## Использование квадратных скобок для выбора столбцов

Хотя в Polars поддерживается нотация для срезов и индексации на основе квадратных скобок (`df[...]`), по большей часть такая поддержка призвана обеспечить плавность перехода и совместимость с другими пакетами. Мы не рекомендуем использовать этот синтаксис, поскольку он гораздо менее эффективен по сравнению с синтаксисом на основе выражений (`pl.col()`). Использование `pl.col()` развязывает оптимизатору Polars руки, позволяя распараллеливать и всячески повышать эффективность выполняемых запро-

сов, особенно при использовании ленивого API. В главе 10 мы уже подробно говорили о том, что нужно сделать, чтобы отойти от нотации с квадратными скобками.

Если вам приходится задумываться о позиции столбца в датафрейме, вместо того чтобы положиться на API в вопросе того, с каким столбцом вы будете работать, значит, вы сбились с правильного пути.

## Неправильное использование метода collect()

Ленивый API в Polars спроектирован с прицелом на максимальную эффективность и быстродействие. При его использовании движок собирает вместе все операции, которые вы планируете выполнить, оптимизирует план выполнения запроса на основе имеющихся у него сведений о данных и лишь затем выполняет сам запрос. Именно в момент вызова метода `collect()` движок запускает на выполнение построенный план и начинает читать данные с диска. Однако при повторном использовании одного и того же ленивого датафрейма вы можете сэкономить немало времени, если не будете заново вызывать метод `collect()` всякий раз, когда вам нужно воспользоваться данными в нем, поскольку в этом случае датафрейм каждый раз будет пересобираться заново. Это может понадобиться, например, если вам необходимо получить два разных набора данных, скажем обучающую и тестовую выборки для метода машинного обучения, или разбить набор данных по поставщикам, как показано ниже:

```
%%time
taxis = pl.scan_parquet("data/taxi/yellow_tripdata_*.parquet")
vendor0 = taxis.filter(pl.col("VendorID") == 0).collect()
vendor1 = taxis.filter(pl.col("VendorID") == 1).collect()
```

Вывод:

```
CPU times: user 5.54 s, sys: 5.3 s, total: 10.8 s
Wall time: 3.62 s
```

Здесь данные с диска будут прочитаны дважды: сначала для первого набора, а затем для второго. Было бы эффективнее материализовать считанные данные, а затем дважды применить к ним фильтрацию следующим образом:

```
%%time
taxis = pl.scan_parquet("data/taxi/yellow_tripdata_*.parquet")
vendors = taxis.filter(pl.col("VendorID").is_in([0, 1])).collect()
vendor0 = vendors.filter(pl.col("VendorID") == 0)
vendor1 = vendors.filter(pl.col("VendorID") == 1)
```

Вывод:

```
CPU times: user 5.82 s, sys: 4.5 s, total: 10.3 s
Wall time: 3.18 s
```

Как видите, это привело к небольшому ускорению. Но это вопрос, тесно связанный с операциями ввода/вывода, так что при работе с большими данными быстродействие может увеличиться существенно.

Также ошибкой является избыточное использование метода `collect()`. Вызывая этот метод применительно к ленивому датафрейму, вы отказываетесь от всех преимуществ ленивой парадигмы вычислений и заставляете движок читать данные и вычислять их без использования оптимизаций. С таким же успехом вы могли бы работать и в рамках жадной парадигмы.

Всегда обращайте внимание на то, когда именно вы вызываете метод `collect()`, поскольку прибегать к нему рекомендуется только тогда, когда вам действительно нужно поместить данные в память.

## Использование кода на Python в запросах Polars

Библиотека Polars спроектирована так, чтобы большая часть кода выполнялась под капотом на языке Rust с использованием параллельных вычислений и скоростью, не сравнимой с Python. При использовании кода на Python в запросах Polars вы вынуждаете Polars переключаться на Python, что неизбежно приводит к замедлению запросов.

Несмотря на то что методы вроде `GroupBy.agg()` и `GroupBy.map_*()` позволяют использовать код на языке Python, вы должны быть с этим осторожны. Всегда обращайтесь к документации API, чтобы отыскать способы достижения нужных вам результатов без использования Python. Кроме того, вы всегда можете обратиться за помощью к отзывчивому сообществу Polars, а также задать интересующие вас вопросы на сервере Polars Discord. Если в API Polars действительно пока не реализован нужный вам функционал, вы можете написать свою собственную пользовательскую функцию на Rust, которая практически гарантированно будет превосходить в скорости аналогичный код на языке Python. Бывает, что вам не избежать использования кода на Python, но вы должны рассматривать это как исключительно крайнюю меру.

# Заключение

В этой главе книги вы узнали, что:

- Polars состоит из нескольких слоев, включая предметно-ориентированный язык, на котором вы формулируете свои запросы, промежуточное представление, содержащее план выполнения запроса с метаданными, оптимизатор для тонкой настройки плана и различные движки для выполнения запроса;
- в основе Polars лежит колоночный формат хранения данных Apache Arrow, хорошо подходящий для аналитических запросов;

- многопоточные вычисления и операции SIMD позволяют Polars извлечь максимум возможного из современного аппаратного обеспечения;
- хранение строковых данных в Polars оптимизировано при помощи макетов представления и буферов данных;
- в Polars применяется масса оптимизационных техник, включая раннее выполнение предикатов, раннее выполнение проекций и раннее осуществление срезов;
- в Polars присутствуют встроенные инструменты профилирования, с помощью которых можно определять узкие места с точки зрения быстродействия;
- Polars содержит набор функций для тестирования, предназначенных для сравнения датафреймов и объектов Series с возможностью поиска точных и приблизительных соответствий;
- при работе с Polars вам следует избегать использования распространенных антипаттернов, таких как злоупотребление квадратными скобками при отборе столбцов, неправильное использование метода `collect()` и избыточное применение кода на Python в сценариях Polars.

Поздравляем! Теперь вы обладаете всеми необходимыми знаниями, чтобы использовать библиотеку Polars на пределе ее немалых возможностей в процессе анализа данных. Но это не конец, а лишь начало. Polars развивается с каждым днем, и вам следует делать то же самое.

Вливайтесь в сообщество Polars и почаще посещайте сервер Discord (*https://discord.com/invite/4qf7UVDZmd*), Stack Overflow (*https://stackoverflow.com/questions/tagged/python-polars*) и репозиторий на Github (*https://github.com/pola-rs/polars*). Делитесь мыслями, задавайте вопросы, отвечайте на вопросы других и вносите свой вклад в развитие экосистемы Polars!

За обновлениями, новыми техниками и дополнительными ресурсами вы можете обратиться на сайт книги по адресу *https://polarsguide.com*.

С Polars процесс анализа данных для вас будет ограничен только вашими желаниями. Так идите вперед и разрабатывайте решения на основе данных следующего поколения!

# Приложение

# Ускорение Polars с помощью графического процессора

Философия Polars частично базируется на том, чтобы по возможности задействовать в процессе работы весь вычислительный потенциал компьютера, состоящий из ядер центрального процессора, а также ядер графического процессора.

В видеокартах NVIDIA, к примеру, используются ядра, в которых применена технология *программно-аппаратной архитектуры параллельных вычислений* (Compute Unified Device Architecture – *CUDA*), – так называемые CUDA-ядра. CUDA представляет собой проприетарную платформу параллельных вычислений, использующую графический процессор (GPU) для ускорения вычислений общего назначения. В отличие от центральных процессоров, в которых, как правило, присутствует не более 20 ядер, современные графические процессоры могут насчитывать более 15 тыс ядер.

И хотя ядра графического процессора по своему вычислительному потенциалу значительно уступают ядрам центрального процессора и предназначены для относительно простых операций, эти операции могут выполняться параллельно, что является большим подспорьем для ускорения вычислений. Если вы можете разбить свою сложную задачу на большое количество простых операций, многоядерная архитектура графического процессора как нельзя лучше может подойти для ее решения.

В этом приложении мы будем обсуждать способы ускорения вычислений в Polars за счет потенциала графического процессора.

В частности, вы узнаете:

- как установить и использовать движок для работы с графическим процессором;
- о поддерживаемых и неподдерживаемых возможностях GPU;
- о проведении эталонного тестирования с помощью GPU;
- в каких сценариях использование ядер графического процессора может дать наибольший прирост производительности;
- о будущих планах Polars при работе с GPU.

# NVIDIA RAPIDS

В компании NVIDIA проект с названием *RAPIDS* описывается как коллекция программных пакетов с открытым исходным кодом и API, дающая возможность выполнять сквозные конвейеры обработки и анализа данных целиком на графических процессорах NVIDIA с использованием знакомых API PyData. Одним из таких пакетов является *cuDF*, предназначенный для работы с датафреймами в Python на графическом процессоре и обладающий интерфейсом, схожим с pandas. Разработчики pandas могут ускорить свои операции по работе с датафреймами, просто импортировав пакет cuDF, при этом они могут практически не менять свой код[1].

А теперь хорошие новости – практически все то же самое доступно вам и в Polars!

Как вы знаете, библиотека Polars славится своей скоростью не только благодаря своей реализации на языке Rust, но и по причине использования оптимизированного движка запросов. Оптимизатор сам определяет, какие операции необходимо выполнить и в каком именно порядке, а самое главное его достоинство состоит в способности отказаться от выполнения избыточных операций.

Однако, в отличие от pandas, в Polars нельзя просто переключить вычисления на другой пакет путем импорта из-за роли движка запросов. Для преодоления этого неудобства Polars и NVIDIA объединили усилия и разработали новый движок GPU, который бесшовно интегрируется в Polars и позволяет извлекать из ресурсов графического процессора все возможное.

Вот как это работает. Движок GPU был разработан совместно командами Polars и RAPIDS и имеет в своей основе пакет *libcudf* – тот же самый, на котором базируется библиотека cuDF. А поскольку этот движок интегрирован в Polars, он способен получать от оптимизатора Polars исправленный план выполнения запроса (см. главу 18) и запускать его на ядрах графического процессора. Это позволяет Polars совмещать использование преимуществ графического процессора посредством пакета libcudf с эффективностью оптимизации плана выполнения запросов.

Если та или иная операция не может быть выполнена на GPU, она автоматически перенаправляется в центральный процессор, что обеспечивает бесперебойность вычислительного процесса. При этом вы можете изменить такое поведение путем конфигурации параметров, о чем мы поговорим далее в этой главе. Единственное изменение, которое вы при этом должны будете внести в свой код, состоит в выборе движка при сборе результатов из ленивого датафрейма. К примеру, если ваш ленивый датафрейм содержится в переменной `lf`, вы просто должны будете изменить строку

```
lf.collect()
```

[1] См. документацию по RAPIDS для активации cuDF в pandas по адресу *https://docs.rapids.ai/api/cudf/nightly/cudf_pandas/usage*.

на

```
lf.collect(engine="gpu")
```

# Установка движка GPU

Чтобы воспользоваться потенциалом движка графического процессора в Polars, ваша система должна отвечать следующим требованиям:

- графическая карта NVIDIA на основе архитектуры Volta и позднее;
- CUDA версии 11 или 12;
- Linux или подсистема Windows для Linux (Windows Subsystem for Linux) версии 2 (WSL2).

В дополнение к этим аппаратным и программным требованиям вам необходимо установить несколько зависимостей. В этом разделе мы пройдем по всему циклу установки, который разобьем на шесть шагов.

Шаги 1 и 2 вам понадобится выполнить, только если вы работаете в среде Windows. Так что если у вас Linux, вы можете сразу переходить к шагу 3. К сожалению, ускорение за счет графического процессора в Polars не поддерживается для операционной системы macOS.

В инструкции предполагается, что у вас стоит Ubuntu 24.04 и CUDA версии 12, но она подойдет и для других дистрибутивов Linux, если следовать приведенным уточнениям в соответствующих разделах.

## Шаг 1: установка WSL2 на Windows

Если вы работаете под Windows, то можете запустить вычисления в Polars на графическом процессоре GPU с использованием подсистемы Windows для Linux версии 2 (WSL2). WSL2 позволяет запускать полноценное ядро Linux на Windows, что дает возможность запускать команды и приложения Linux на компьютере под управлением Windows. Первое, что мы сделаем, – это научимся устанавливать WSL2 на Windows, после чего инсталлируем оболочку Ubuntu Linux.

В Windows 11 подсистема WSL2 уже встроена. Для проверки вы можете выполнить следующие действия.

1. Нажмите на клавишу **Windows**.
2. Введите в поиске **PowerShell**.
3. Щелкните правой кнопкой мыши по найденному приложению PowerShell и выберите пункт запуска от имени администратора.

В открывшемся окне PowerShell введите следующую команду:

```
PS > wsl --version
```

Вывод:

```
WSL version: 2.2.4.0
Kernel version: 5.15.153.1-2
WSLg version: 1.0.61
MSRDC version: 1.2.5326
Direct3D version: 1.611.1-81528511
DXCore version: 10.0.26091.1-240325-1447.ge-release
Windows version: 10.0.26100.2033
```

Если в первой строке вывода вы увидите версию WSL, начинающуюся с двойки, значит, вам больше ничего делать не нужно. Но никогда не мешает обновиться до последней доступной версии. Для этого выполните следующую команду:

```
PS > wsl --update
```

Если же у вас установлен WSL первой версии, вам необходимо будет обновить его до второй версии, воспользовавшись инструкциями по адресу *https://learn.microsoft.com/en-us/windows/wsl/install*.

## Шаг 2: установка Ubuntu Linux на WSL2

После установки WSL2 вам необходимо обзавестись дистрибутивом Linux. Версией по умолчанию, которую мы также рекомендуем, является оболочка Ubuntu 24.04.

1. Выполните показанную ниже команду в PowerShell:

   ```
   PS > wsl --install
   ```

2. Перезагрузите компьютер для завершения установки.

После перезагрузки в меню **Пуск** будет доступен пункт **Ubuntu**. Это терминал, в котором вы можете вводить команды Linux из следующего раздела.

**Подключение среды разработки к WSL2**

Современные среды разработки, такие как VS Code и IntelliJ IDEA, обладают возможностью подключения к WSL2 для удобств разработки. Для подключения VS Code к WSL2 следуйте официальной инструкции от Microsoft по адресу *https://learn.microsoft.com/en-us/windows/wsl/tutorials/wsl-vscode*. Инструкция для среды IntelliJ IDEA находится по адресу *https://www.jetbrains.com/help/idea/how-to-use-wsl-development-environment-in-product.html*.

После настройки среды разработки и установки WSL2 вы можете переходить к шагу 3. Для этого запустите приложение WSL2 вручную или подключите свою среду разработки к WSL2.

## Шаг 3: установка необходимых пакетов в Ubuntu Linux

Перед установкой пакета для работы с GPU вам необходимо озаботиться установкой вспомогательных пакетов в Ubuntu Linux. Они потребуются для сборки и запуска необходимых компонент. Для начала обновите индексную информацию о пакетах с помощью следующей команды:

```
$ sudo apt update
```

Обратите внимание на строку приглашения в виде знака доллара ($). Это означает, что приведенные здесь команды необходимо вводить в терминале Linux, а не в PowerShell. Затем обновите все пакеты до актуальных версий следующей командой:

```
$ sudo apt upgrade -y
```

Наконец, установите пакет build-essential, включающий в себя компиляторы и иные инструменты, которые понадобятся вам в дальнейшем:

```
$ sudo apt install build-essential -y
```

## Шаг 4: установка набора инструментов CUDA

*Набор инструментов CUDA* (CUDA Toolkit) представляет собой *пакет средств разработки программного обеспечения* (software development kit – SDK), созданный NVIDIA для сборки и запуска приложений, использующих технологию CUDA, к которым относится и движок GPU для Polars. Для установки набора инструментов CUDA вам сперва нужно узнать, какая версия CUDA у вас установлена. Это поможет вам настроить правильные зависимости.

1. Выполните следующую команду для проверки версии CUDA:

   ```
   $ nvidia-smi | grep "CUDA Version"
   ```

   В зависимости от установленной версии вы увидите вывод 11.X или 12.X. Как мы упоминали выше, приведенные здесь инструкции соответствуют 12-й версии CUDA. Если у вас 11-я версия, которую Polars не поддерживает, обратитесь к инструкциям по адресу *https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html*.

   Сейчас мы последуем инструкциям по установке со страницы по адресу *https://developer.nvidia.com/cuda-downloads*.
2. Запустите следующие команды в терминале Linux (обратите внимание, что адрес после команды `wget` должен располагаться на одной строке):

   ```
   $ wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/x86_64/
   cuda-keyring_1.1-1_all.deb
   ```

```
$ sudo dpkg -i cuda-keyring_1.1-1_all.deb
$ sudo apt update
$ sudo apt -y install cuda-toolkit-12-6
```

В результате будет установлен пакет с набором ключей CUDA, обновлен список ваших пакетов и установлен набор инструментов CUDA.

## Шаг 5: установка зависимостей Python

Теперь, когда мы установили все вспомогательные пакеты, можно приступать к настройке зависимостей в Python. На момент написания книги зависимости доступны для Python вплоть до версии 3.12. Вы можете обратиться к документации RAPIDS по адресу *https://docs.rapids.ai/install* за актуальной информацией.

Установите необходимые зависимости, запустив следующую инструкцию из командной строки Linux:

```
$ uv pip install polars[gpu]
```

## Шаг 6: проверка установки

Запустите следующий фрагмент кода для проверки успешности установки. Вы можете ввести этот код, как в интерпретаторе Python, прямо в терминале Linux (для этого нужно предварительно запустить сам интерпретатор командой `python`), так и в Jupyter Notebook:

```
import polars as pl

pl.LazyFrame({"x": [1, 2, 3]}).collect(engine=pl.GPUEngine(raise_on_fail=True))
```

Вывод:

```
shape: (3, 1)
┌─────┐
│ x   │
│ --- │
│ i64 │
╞═════╡
│ 1   │
│ 2   │
│ 3   │
└─────┘
```

Здесь мы передали методу `collect()` аргумент `engine` с функцией `pl.GPUEngine()` и параметром `raise_on_fail`, установленным в `True`. Это значит, что в случае безуспешной попытки выполнить представленную инструкцию силами графического процессора будет выдана ошибка вместо передачи инструкции движку центрального процессора. Это позволит вам проверить,

успешно ли был установлен движок GPU. О конфигурировании этого движка мы поговорим в следующем разделе.

В случае возникновения ошибки обратитесь к документации RAPIDS по адресу *https://docs.rapids.ai/install/#troubleshooting*.

# Использование движка GPU в Polars

После установки движка GPU использовать его можно очень просто. Любой вызов метода `lf.collect()` можно изменить, передав ему нужное значение в параметре `engine`. Для выполнения запроса силами графического процессора вы можете воспользоваться инструкцией `lf.collect(engine="gpu")`. Все просто.

## Настройка движка GPU

Если вы хотите перехватить контроль над конфигурацией движка GPU, вы можете воспользоваться функцией *`pl.GPUEngine()`*. Эта функция принимает три следующих аргумента:

- `device`: если у вас установлено несколько графических карт в системе, вы можете выбрать ту из них, которую необходимо использовать. Это целочисленный аргумент, принимающий идентификатор устройства. Узнать идентификаторы графических устройств можно, запустив команду `nvidia-smi` в терминале Linux;
- `raise_on_fail`: по умолчанию в случае неудачи выполнения инструкции силами графического процессора она молча передается центральному процессору. Если передать этому аргументу значение `True`, вместо этого будет генерироваться ошибка;
- `memory_resource`: с помощью этого аргумента можно задать способ выделения памяти механизмом RAPIDS Memory Manager (RMM). Мы не будем глубоко погружаться в эту тему, но если вам интересно, вы можете обратиться к официальной документации RMM по адресу *https://github.com/rapidsai/rmm?tab=readme-ov-file#memoryresource-objects*.

**Почему же, почему?**

При запуске Polars в режиме расширенного вывода посредством инструкции `config pl.Config().set_verbose(True)` вы будете видеть не только уведомление о том, что выполнение запроса было передано с GPU на CPU, но и причину этого.

К примеру, на момент написания книги объединения на основе неравенств не поддерживались графическим процессором, и при попытке выполнить их на GPU мы получили бы следующее предупреждение:

```
PerformanceWarning: Query execution with GPU not supported,
reason: <class 'NotImplementedError'>: IEJoin
```

## Неподдерживаемые возможности

На протяжении всей книги мы писали о богатых возможностях API Polars. Но в движке GPU реализованы далеко не все функции и методы, присутствующие в API.

Первое, что нужно упомянуть, – это то, что на графическом процессоре не поддерживается ни одна операция из жадного API, вместо этого вам необходимо использовать ленивый интерфейс. Кроме того, на момент написания книги на GPU не поддерживаются следующие операции:

- чтение данных из JSON, Excel и баз данных;
- операции с типами данных `Categorical`, `Struct` и `List`;
- пользовательские функции (UDF);
- скользящие агрегации и оконные функции;
- передискретизация временных рядов и часовые пояса;
- сворачивание данных и горизонтальные агрегации.

# Эталонное тестирование движка GPU в Polars

Мы посмотрели, как можно настраивать движок GPU, как необходимо его использовать и для каких операций он, увы, неприменим. В этом разделе мы проведем сравнительный анализ движка GPU с другими движками в Polars, а также с другими пакетами для работы с датафреймами, как использующими мощности графического процессора, так и обходящимися услугами CPU, на разных объемах данных.

## Решения

Сравнительные тесты были запущены с использованием версий библиотек, перечисленных в табл. П-1.

***Таблица П-1. Используемые библиотеки и их версии***

| Библиотека | Версия |
|---|---|
| cuDF | 24.12.00 |
| Dask | 2024.12.1 |
| DuckDB | 1.1.3 |
| pandas | 2.2.3 |
| Polars | 1.20.0 |
| Polars с GPU | 1.14.0 |
| PySpark | 3.5.4 |

## Запросы и данные

Проводить эталонное тестирование непросто. Для достижения максимально объективных результатов команда разработчиков Polars создала свой собственный тестовый репозиторий *Polars Decision Support* (PDS) на основе теста TPC-H. *TPC-H* – это эталонный тест для оценки производительности систем поддержки принятия решений. Подробнее о нем вы можете узнать на официальной странице по адресу *https://www.tpc.org/tpch*. Мы создали ветку этого репозитория (*https://github.com/TNieuwdorp/polars-benchmark*), чтобы можно было выполнять проверки в нашем тестовом окружении с небольшими изменениями, и реализовали отсутствующие запросы для pandas.

Тест состоит из 22 запросов, запускаемых на наборе данных. Наборы данных генерируются самим тестом и хранятся в файлах Parquet. Дополнительно запросы запускаются на других объемах данных, что управляется коэффициентом масштабирования. За эталон берется время самого запроса, включая время на чтение данных с диска. Импортирование пакетов не включается в замер времени.

Чтобы вы понимали, о каких запросах идет речь, взгляните на запрос под номером 15:

```
%%time
lineitem = pl.scan_parquet("data/benchmark/lineitem.parquet") ❶
supplier = pl.scan_parquet("data/benchmark/supplier.parquet")

var1 = pl.date(1996, 1, 1)
var2 = pl.date(1996, 4, 1)

revenue = (
    lineitem.filter(pl.col("l_shipdate").is_between(var1, var2, closed="left"))
    .group_by("l_suppkey")
    .agg(
        (pl.col("l_extendedprice") * (1 - pl.col("l_discount")))
        .sum()
        .alias("total_revenue")
    )
    .select(pl.col("l_suppkey").alias("supplier_no"), pl.col("total_revenue"))
)

query_15 = (
    supplier.join(revenue, left_on="s_suppkey", right_on="supplier_no")
    .filter(pl.col("total_revenue") == pl.col("total_revenue").max())
    .with_columns(pl.col("total_revenue").round(2))
    .select("s_suppkey", "s_name", "s_address", "s_phone", "total_revenue")
    .sort("s_suppkey")
)

query_15.collect(engine="cpu") ❷
```

❶ Операции чтения данных из файла включаются в замер времени.

❷ Движок по умолчанию – cpu. Для использования движка графического процессора необходимо поменять это значение на gpu.

Вывод:

```
CPU times: user 17.7 ms, sys: 16.5 ms, total: 34.2 ms
Wall time: 10.9 ms
shape: (1, 5)
```

| s_suppkey | s_name | s_address | s_phone | total_revenue |
|---|---|---|---|---|
| i64 | str | str | str | f64 |
| 677 | Supplier#000000677 | 8mhrffG7D2WJBSQbOGst… | 23-290-639-3315 | 1.6144e6 |

# Оборудование

Мы запускали тесты на рабочей станции Dell Precision 5860 Tower, спецификацию которой можно посмотреть по адресу *https://www.dell.com/en-us/shop/desktop-computers/precision-5860-tower/spd/precision-5860-workstation*. Также мы перечислили основные характеристики в табл. П-2.

**На плечах гигантов**

Мы безмерно признательны компаниям NVIDIA и Dell Technologies за то, что они откликнулись на наше желание провести подобный сравнительный анализ и предоставили все необходимое для этого аппаратное обеспечение. В разделе с благодарностями в начале книги мы рассказали об этом подробнее.

***Таблица П-2. Спецификация тестовой системы***

| Характеристика | Значение |
|---|---|
| Название | Dell Precision 5860 Tower |
| Центральный процессор | Intel Xeon w7-2495X 2.50 GHz |
| Оперативная память | 128 GB (8 x 16 GB), 4,800 MT/s DIMM, DDR5 |
| Операционная система | Windows 11 Pro for Workstations |
| Версия | Version 24H2,<br>OS build 26100.2033,<br>Feature Experience Pack 1000.26100.23.0 |

В процессе тестирования мы использовали несколько видеокарт для проверки разных возможностей (см. табл. П-3). Одна из этих видеокарт (A1000) реализована на основе архитектуры Ampere, остальные – на основе Ada Lovelace. На момент написания книги архитектура Ampere уже считалась устаревшей, а ей на смену пришла архитектура Ada Lovelace. Быстродействие приведено с одинарной точностью.

*Таблица П-3. Спецификации использованных видеокарт*

| | A1000 | 2000 Ada | 4000 Ada | 5000 Ada | 6000 Ada |
|---|---|---|---|---|---|
| **Архитектура** | Ampere | Ada Lovelace | Ada Lovelace | Ada Lovelace | Ada Lovelace |
| **CUDA-ядра** | 2304 | 2816 | 6144 | 12 800 | 18 176 |
| **Производительность (TFLOPS)** | 6.7 | 12.0 | 26.7 | 65.3 | 91.1 |
| **Видеопамять (Гб)** | 8 | 16 | 20 | 32 | 48 |

Код тестирования доступен в репозитории по адресу *https://github.com/TNieuwdorp/polars-benchmark*. Наборы данных – на основе теста TPC-H.

Каждый набор тестово прогонялся с использованием коэффициентов масштабирования 0.1, 1.0, 5.0, 10.0, 20.0, 35.0 и 50.0. По сути, это соответствует размеру базы данных в Гб, если бы она хранилась в файле в формате CSV. Но поскольку фактически данные у нас хранятся в формате Parquet, они занимают меньше места. При загрузке данных в память их размер увеличивается, но мы не учитываем эти размеры в тесте. Тесты запускали по пять раз для каждого коэффициента масштабирования с целью получения достоверных результатов. Минимальные и максимальные значения исключались, чтобы избежать учета выбросов, а оставшиеся мы усреднили для получения итогового результата.

# Результаты и обсуждение

Первоначально мы также включили в программу тестирования библиотеку Modin с движком Ray, но ее использование приводило к повышенному расходу дискового пространства – до 1 Тб, что было чересчур для нашего тестового окружения. В результате мы исключили этот пакет из общих результатов.

На высоких значениях коэффициента масштабирования мы проверяли только самые высокопроизводительные решения на основе Polars, DuckDB и cuDF, чтобы не тратить много времени.

В процессе тестирования мы обнаружили два запроса (11 и 22) с операциями объединения на основе неравенства, не поддерживаемыми движком GPU.

Проведенное тестирование позволило ответить на многие важные вопросы, о чем мы будем упоминать отдельно.

## *Движок Polars GPU против Polars CPU*

Во-первых, посмотрим на результаты сравнительного анализа движков центрального и графического процессоров в Polars.

На рис. П-1 видно, что движок GPU превосходит движок CPU в большинстве случаев. Но нельзя не заметить, что при использовании низкого коэффициента масштабирования быстрее оказался центральный процессор. Причина в накладных расходах на перенос данных в графический процессор. Но с ростом объемов преимущество движка GPU становится неоспоримым.

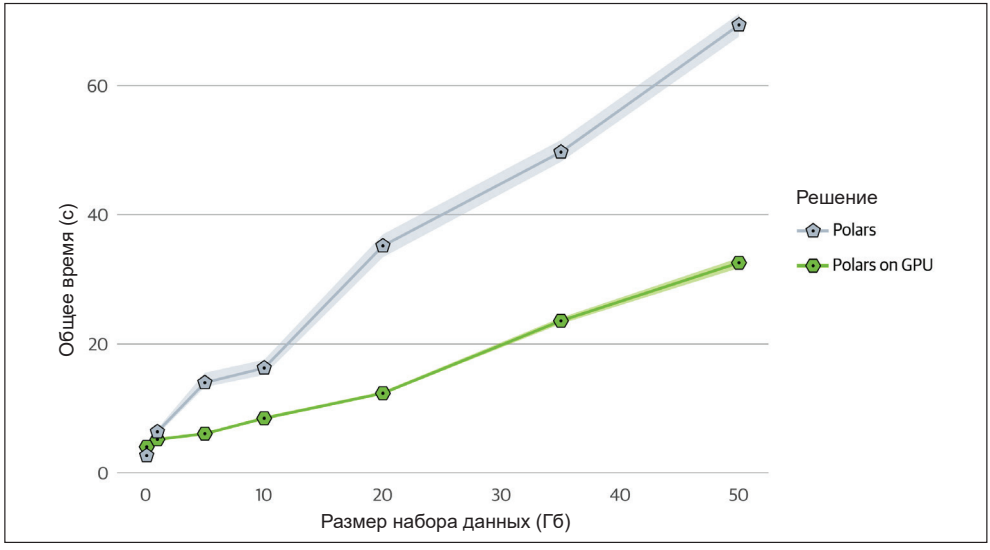



**Рис. П-1** ❖ Сравнительный анализ движков Polars GPU и Polars CPU для разных объемов данных на видеокарте NVIDIA 6000 Ada Lovelace

Как видно на рис. П-2 с относительным приростом, начиная с 5 Гб объема данных движок Polars GPU практически вдвое превосходит движок Polars CPU. Из этого можно сделать вывод о том, что графический процессор лучше задействовать при работе с большими данными.

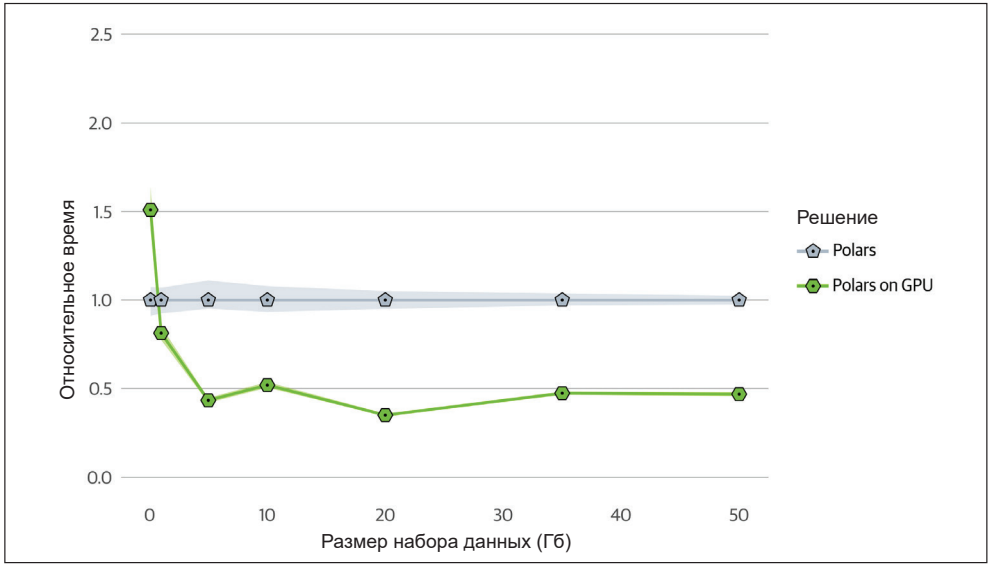



**Рис. П-2** ❖ Относительный прирост производительности движка Polars GPU в сравнении с Polars CPU для разных объемов данных на видеокарте NVIDIA 6000 Ada Lovelace

## *Быстродействие на других вариантах аппаратного обеспечения*

Предыдущие результаты были показаны на мощной видеокарте RTX 6000 Ada GPU.

А как себя покажут движки на другом железе? На рис. П-3 видно, какое сильное влияние оказывает выбранная видеокарта.

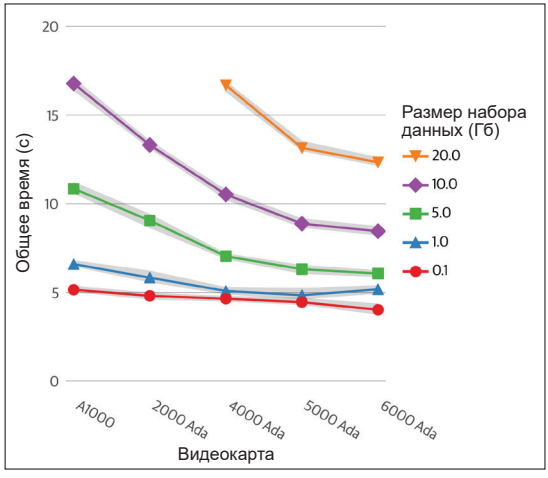



**Рис. П-3** ❖ Сравнительный анализ разных видеокарт в зависимости от объема обрабатываемых данных

Еще более интересным может быть сравнение в терафлопсах (TFLOPS), показанное на рис. П-4.

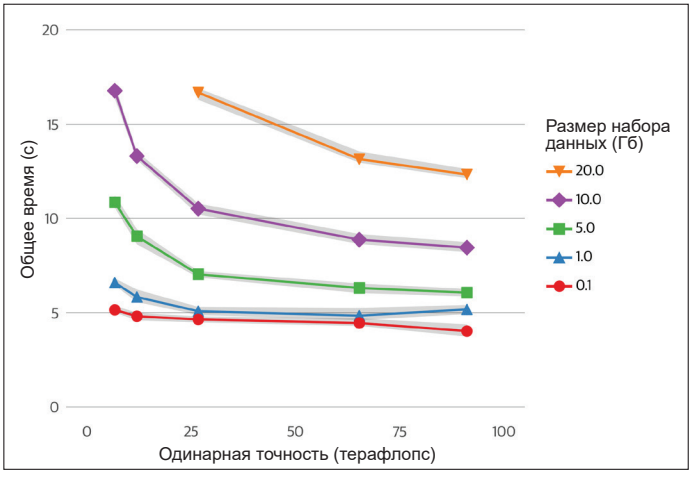



**Рис. П-4** ❖ Сравнительный анализ в терафлопсах в зависимости от объема обрабатываемых данных

Здесь мы видим, что с увеличением вычислительной мощности прирост производительности начинает снижаться. Этот фактор убывающей отдачи очень интересен сам по себе и говорит о нелинейности роста быстродействия движка GPU в зависимости от производительности.

На всех проверенных нами видеокартах, даже на карте предыдущего поколения, A1000, движок GPU выигрывал в скорости у движка CPU при объемах данных от 5 Гб, что видно на рис. П-5.

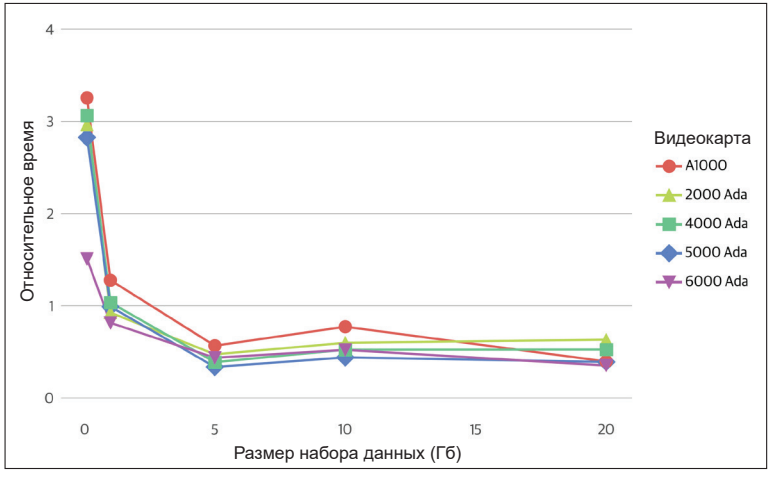



**Рис. П-5** ❖ Сравнительный анализ движка GPU и движка CPU на разных видеокартах

## *Сравнение движка Polars GPU с другими пакетами*

Теперь, когда мы завершили сравнительный анализ в рамках Polars, пришло время взглянуть на другие пакеты для работы с данными.

На рис. П-6 показано, что оба движка, задействующих GPU, опережают движки, работающие исключительно с использованием мощностей центрального процессора.

На рис. П-7 показано, что пакет cuDF также страдает от накладных расходов при работе с данными небольшого объема.

## *Влияние оптимизатора Polars*

Еще один любопытный вопрос, ответ на который нам хотелось узнать, заключается в том, какой вклад в быстродействие вносит оптимизатор Polars. Поскольку пакет cuDF и движок GPU в Polars базируются на одной и той же библиотеке libcudf, единственным отличием между ними, по сути, является вклад оптимизатора Polars. Оптимизатор может значительно менять план выполнения запросов, добавлять и исключать операторы, что оказывает влияние на производительность.

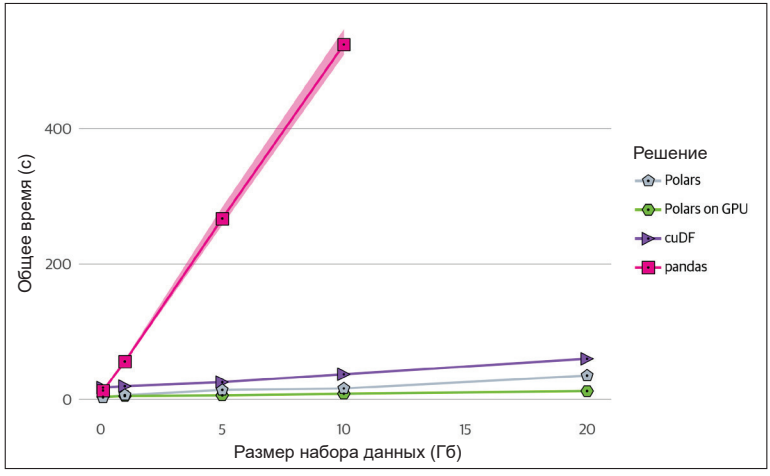



**Рис. П-6** ❖ Сравнительный анализ движков Polars с аналогами в pandas и cuDF

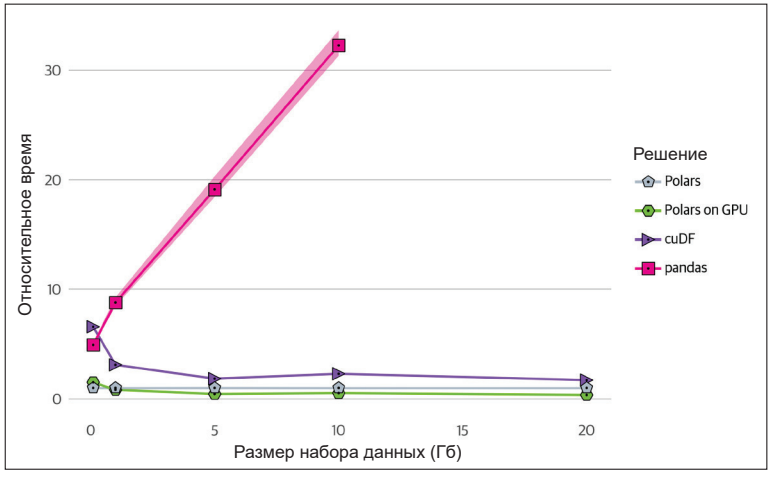



**Рис. П-7** ❖ Эффект накладных расходов, связанных с перемещением данных в память графического процессора

На рис. П-8 видно, что оптимизатор Polars действительно не зря ест свой хлеб. Более чем четырехкратное превосходство в быстродействии в сравнении с пакетом cuDF, также активно использующим мощности графического процессора, говорит само за себя.

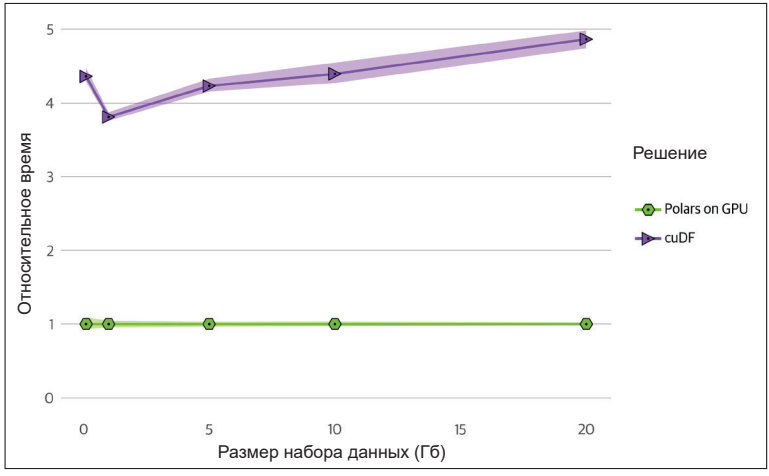



**Рис. П-8** ❖ Сравнительный анализ движка GPU в Polars с пакетом cuDF

## *Выводы*

Проведенное эталонное тестирование показывает, что привлечение к работе графического процессора бывает наиболее оправдано при решении сложных вычислительных задач. Таким образом, операции вроде `join()`, `group_by()` и `GroupBy.agg()` могут быть первыми кандидатами на выполнение с использованием движка GPU. Запросы, связанные с операциями ввода-вывода и чтением с диска, не сильно приобретут от задействования графического процессора. Более того, такие операции могут даже замедлиться по причине накладных расходов на перемещение данных в память GPU. Кроме того, на эффективность привлечения мощностей графического процессора также влияет объем обрабатываемых данных. Мы видели, что разница проявляется начиная примерно с 5 Гб – это также связано с накладными расходами на использование движка GPU. При этом рост эффективности с увеличением вычислительных мощностей графической карты не является линейным и в какой-то момент сильно замедляется.

Также важно помнить, что графические процессоры часто бывают гораздо сильнее ограничены с точки зрения памяти в сравнении с центральными процессорами, так что обработка очень больших данных с использованием GPU может завершиться неудачей.

В целом можно отметить, что превосходство движка GPU над движком CPU в Polars в среднем является двукратным, а другие пакеты для обработки данных движок GPU в Polars оставляет далеко позади.

# Будущие планы по использованию графического процессора в Polars

Важно заметить, что использование движка GPU в Polars по-прежнему находится на стадии бета-тестирования и в данный момент разработчики трудятся над расширением его потенциала. А вот какие у них планы на будущее:

- выполнение на CPU операций, которые пока не поддерживаются GPU;
- одновременное выполнение подзапросов на GPU и CPU для повышения быстродействия;
- параллельное выполнение на нескольких GPU;
- использование unified-памяти CUDA с целью оптимизации управления памятью и расширения возможностей движка в отношении объема обрабатываемых данных;
- использование потокового движка для обработки больших данных, не помещающихся в память графического процессора.

Последние два пункта в планах могут нивелировать ограничения движка GPU, связанные с памятью, что будет означать новый прорыв в использовании этой архитектуры вычислений.

# Заключение

В этом приложении мы обсудили следующие темы:

- возможности использования движка GPU в Polars;
- процесс установки графического движка;
- особенности использования движка GPU;
- неподдерживаемые возможности движка GPU;
- эталонное тестирование всех видов движков в Polars с другими пакетами для обработки данных;
- результаты и выводы эталонного тестирования;
- планы Polars по расширению использования графического процессора в области вычислений.

Теперь вы понимаете все аспекты использования движка GPU в Polars и можете задействовать его в своих проектах по мере необходимости.

# Предметный указатель

## F



## G

## R



## S

Йерун Янссенс, Тейс Ньюдорп

**Python Polars: подробное руководство**

Главный редактор *Мовчан Д. А.*
Зам. главного редактора *Яценков В. С.*
editor@dmkpress.com
Перевод *Гинько А. Ю.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура PT Serif. Печать цифровая.
Усл. печ. л. 40,79. Тираж 100 экз.

«Polars – восходящая звезда на горизонте экосистемы обработки и анализа данных в Python. Йерун и Тейс очень вовремя представили общественности свою книгу, позволяющую постичь все нюансы и преимущества этой библиотеки».

Уэс Маккини, создатель pandas и главный архитектор Posit PB

# Python Polars: подробное руководство

Откройте для себя Polars – удивительно эффективную библиотеку для преобразования, анализа и визуализации данных. В этом подробном руководстве продемонстрированы все ключевые возможности Polars и показано практическое применение в задачах, связанных с обработкой и исследованием данных, построением конвейеров и многим другим.

Независимо от квалификации, вы быстро освоите интуитивно понятный API Polars. На GitHub доступно множество примеров работы с реальными данными, облегчающими изучение библиотеки.

Прочитав эту книгу, вы:

- научитесь эффективно обрабатывать данные, представленные в форматах CSV и Parquet, электронных таблицах и базах данных, а также расположенные в облаке;
- освоите работу со сложными типами данных, включая текст, дату и время, а также вложенные структуры;
- научитесь использовать жадный и ленивый интерфейсы и узнаете, когда их следует применять;
- научитесь визуализировать данные при помощи пакетов Altair, hvPlot, plotnine и Great Tables;
- сможете расширять возможности Polars при помощи собственных функций на Python и плагинов на Rust.

**Йерун Янссенс** – старший инженер по связям с разработчиками в Posit PBC. Ярый приверженец проектов с открытым исходным кодом и любит делиться своими знаниями. Автор книги Data Science at the Command Line (O'Reilly). Йерун обладает докторской степенью в области машинного обучения Тилбургского университета и степенью магистра наук в области искусственного интеллекта Маастрихтского университета.

**Тейс Ньюдорп** – главный специалист по обработке и анализу данных в компании Xomnia. Одним из первых начал использовать Polars, внес в эту библиотеку весомый вклад и продолжает использовать ее на ежедневной основе. Тейс увлечен построением инновационных автоматизированных рабочих процессов, позволяющих повысить эффективность труда и облегчить извлечение выводов для принятия стратегических решений.

ISBN 978-6-01140-650-5

9 786011 406505 >

O'REILLY®