**Тема 1/4: Как мы учим?**

**Спринты, теория и проекты**

Наш курс состоит из нескольких тематических блоков. Каждый блок делится на двухнедельные отрезки. Такой отрезок называют **спринтом**.

В течение одного спринта вы изучаете теорию и выполняете задания в онлайн-тренажере. Каждый спринт завершается домашним заданием — самостоятельным **проектом**.

В первую неделю спринта мы рекомендуем пройти теорию и выполнить задания тренажёра, а вторую неделю посвятить практическому проекту и его доработке.

### О проектах и проверке

Каждый проект — это задача, с которой любой разработчик может столкнуться в реальности.

Успешное выполнение проекта зависит

* от технических знаний и навыков;
* от понимания логики построения кода;
* от умения запросить информацию и пояснить свой запрос, от навыков системного мышления, от способности делиться своими результатами с командой — от наличия тех навыков, которые называют soft skills.

После того, как вы отправите проект на проверку, в течение суток ревьюер пришлёт комментарии и правки. Проект нужно будет доработать и отправить его на повторную проверку.

Проверка может состоять из нескольких таких циклов, и когда ревьюер зачтёт все изменения — проект будет принят.

# Дедлайны и каникулы

Проходить теорию и выполнять проекты нужно в определённые сроки. Дату, определяющую последний день такого срока, в Практикуме называют «дедлайн»; у нас есть два типа дедлайнов: мягкие и жёсткие.

## Мягкий дедлайн

Мягкий дедлайн — это окончание спринта. В идеальной ситуации вы приступаете к новому спринту в день его открытия на платформе, а ровно через две недели ревьюер принимает ваш учебный проект.

Если пройти спринт за две недели не получилось —

* предупредите об этом своего куратора, попросите совета и вместе составьте план дальнейших действий;
* задайте вопросы по спринту наставникам и одногруппникам в общих чатах.

При нарушении мягкого дедлайна доступ к материалу следующих спринтов не блокируется.

## Жёсткий дедлайн

Жёсткий дедлайн — это последний день перед каникулами. Каникулы следуют за каждым большим модулем курса.

На протяжении всего курса у вас будет несколько жёстких дедлайнов. Куратор опубликует подробное расписание вашей когорты, с датами дедлайнов и каникул.

В идеальной ситуации вы успеваете пройти все спринты до каникул, а в каникулы отдыхаете. В отдельных случаях куратор может предложить вам доделать оставшиеся задания в каникулы, но для этого вам нужно заранее предупредить куратора об отставании.

При нарушении жёсткого дедлайна студента ждёт академический отпуск. Об академических отпусках мы расскажем в следующем уроке.

## Каникулы

Каждые каникулы продолжаются неделю и подразумевают перерыв в занятиях. В это время вы сможете отдохнуть, изучить предложения о работе и обсудить с одногруппниками возникшие вопросы.

Доступ ко всем материалам сохраняется и во время каникул.

## Плагиат

Мы отчисляем за плагиат.

# Академический отпуск

Академический отпуск предоставляется по другим условиям, если вы B2B-студент или студент программы **«Цифровые профессии»**. За дополнительной информацией обратитесь к куратору когорты.

## Для чего нужен академический отпуск

Обучение на нашем курсе построено на системе дедлайнов: студент должен сдать все открытые проекты к строго определённому сроку. Но бывает так, что уложиться в срок не получается: помимо учёбы у студентов есть семья, основная работа и множество других забот.

Если по каким-то причинам студент вынужден сделать перерыв в учёбе — он может взять академический отпуск.

## Академ можно брать только два раза

Академический отпуск можно брать лишь два раза за время прохождения курса. Каждый отпуск может длиться не больше месяца.

## Перевод в другую когорту без оформления академа

Если до начала основного курса студент не успел пройти бесплатную тему «Основы Python» — он может перевестись в другую когорту, не оформляя академический отпуск.

В такой ситуации студент сперва проходит вводный курс, а затем переводится в ту когорту, которая начнёт обучение в ближайшее время.

## Когда не стоит брать академ

Не стоит брать академический отпуск тем студентам, которые отстали от своей когорты на один спринт: в этой ситуации есть все шансы догнать своих одногруппников. Но об отставании лучше сообщить куратору — он поддержит и посоветует, как ускориться.

## Когда стоит взять академ

Если студент не успел сдать все открытые проекты к жёсткому дедлайну, куратор напишет ему и предложит оформить академический отпуск.

Обычно мы рекомендуем взять академ тем, кто отстал от своей когорты на два спринта и более. В этом случае студенту проще перейти в новую когорту и продолжить учёбу с того урока, на котором он остановился.

Оформить академ мы советуем и тем студентам, которые по каким-то личным причинам не смогут учиться в определённый период. Лучше заранее оформить академический отпуск, чем переживать по поводу отставания от когорты.

## Как правильно оформить академ

Чтобы взять академический отпуск — напишите куратору, на каком спринте вы остановились и когда сможете вернуться к учёбе. Куратор подскажет, к какой когорте вам будет лучше присоединиться после отпуска и пришлёт форму для заявки на оформление отпуска; эту форму нужно будет обязательно заполнить.

До окончания академического отпуска студент остаётся в прежней когорте, у него сохраняются все доступы к урокам и к мессенджеру; во время отпуска можно продолжать обучение и сдавать проекты.

## Не пропадайте после академа

При оформлении академического отпуска поставьте себе напоминание в календаре о дате возвращения из отпуска. Если в течение недели по окончании отпуска студент не свяжется с куратором — мы автоматически оформим второй академ.

## Как вернуться к учебе после академа

В тот день, когда студент выйдет из отпуска, новый куратор подключит студента к чатам новой когорты и сообщит имя нового наставника. При этом ревьюер останется тем же, что и был.

## Если два академа не помогли

Студентам, которые взяли два академических отпуска, но на очередном жёстком дедлайне не сдали проекты, мы предлагаем прервать обучение и оформить возврат денег.

Практика показывает, что для таких студентов нет смысла продолжать обучение: либо курс не приносит им пользы, либо у них нет возможности включить регулярные занятия в своё расписание.

## Про деньги

Академический отпуск не изменит общую стоимость курса.

Во время академического отпуска списания денег продолжатся, ведь академ — это штатная часть обучения (во время отпуска у вас остаётся доступ к платформе и к студенческим чатам, сохраняется возможность работать с командой сопровождения).

### При оплате в рассрочку

Каждый месяц с вашей карты списывается определённая сумма, и она будет списываться до тех пор, пока полная стоимость курса не будет погашена полностью. Во время академического отпуска списания будут продолжаться, общая сумма оплаты и количество платежей не изменятся. Таким образом, при оформлении академического отпуска платежи закончатся раньше, чем завершится обучение.

### При оплате курса целиком

При единоразовой полной оплате курса вся сумма попадает на ваш виртуальный счёт, деньги с него списываются ежедневно. Формула есть в седьмом пункте [оферты](https://yandex.ru/legal/praktikum_offer/), но если вкратце — остаток на этом счёте каждый день уменьшается.

Если вы возьмёте академический отпуск — общая сумма оплаты и количество платежей не изменятся и все деньги с вашего виртуального счёта будут списаны раньше, чем вы закончите курс обучения.

# Наставники и поддержка

Во время учёбы вас будет поддерживать целая команда: куратор, наставники, ревьюеры, сотрудники поддержки. Вы будете общаться с ними в студенческих чатах.

## Куратор

**Куратор** — хранитель организационной стороны программы и её атмосферы.

Он знает, когда будут каникулы, где отвечают на вопросы наставники, что делать, если возникли проблемы со входом в личный кабинет. Он посочувствует, если что-то не получилось, и поможет придумать, что делать, если день рождения бабушки совпадает со сдачей итогового проекта.

Куратор не знает основ программирования, с вопросами по коду вам помогут наставники.

## Наставник

**Наставники** — опытные разработчики и ваши проводники в практическую часть обучения. Они выполняют роль старших коллег на реальной работе. Главная задача наставника — научить студента системному мышлению, умению решать нетривиальные задачи в программировании.

Каждый день, включая выходные, наставники будут отвечать на ваши вопросы в чатах по теории и практике. Каждые две недели наставники проводят вебинары.

В большинстве случаев наставники ответят на ваш вопрос в течение дня. Если после ответа возникли дополнительные вопросы, задавайте их в том же треде.

**Наставники не отвечают в личных сообщениях** на вопросы по заданиям, теории и проектам. Такие вопросы обсуждаются в общем канале, чтобы дискуссия была доступна всем студентам.

Наставники **не** подсказывают готовые решения и **не** консультируют по темам, которых нет в программе.

В ответ на один вопрос студента наставник может задать три своих. Это нормально: при любой возможности мы будем провоцировать вас на самостоятельное исследование.

## Код-ревьюер

**Код-ревьюер** — это опытный разработчик, который будет проверять ваши проекты и играть роль тимлида. Обычно вы будете справляться с проектом за 2-3 итерации «проверка → комментарий ревьюера → внесение правок → проверка...».

Чтобы уточнить комментарии ревьюера, на каждой итерации правок можно написать лишь одно сообщение ему в личку. На реальной работе руководитель не будет отвечать целый день на ваши вопросы, учитесь структурировать их - в одном сообщении сразу уточнять все, что хочется спросить.

Ревью проекта требует времени, но если ревью вашего проекта заняло более суток — обязательно сообщите куратору.

Код-ревьюер отвечает только на вопросы по проектам.

## Образовательная поддержка/Support

**Образовательная поддержка** — это команда быстрого реагирования, которая поможет с любыми вопросами по тренажёру. Для связи с поддержкой кликните по значку чата в правом нижнем углу окна тренажёра.

Сотрудники образовательной поддержки помогут побороть баги, отличить проблемы с кодом от проблем с тренажёром, найти нужные кнопки.

Поддержка не знает, когда у вас каникулы и как нужно решать итоговый проект спринта: на первый вопрос может ответить куратор, а на второй — наставник.

# Как устроен мессенджер Пачка

Комьюнити Практикума базируется в **мессенджере Пачка** — это удобное приложение для личных и групповых чатов, основной канал коммуникации при обучении на нашем курсе.

Работать c мессенджером можно через мобильное или настольное приложение, а можно через браузер — через веб-интерфейс. Браузерная версия лучше и стабильнее, рекомендуем именно её.

При умелом обращении Пачка станет дополнительным инструментом для обучения: правильно заданные вопросы помогут справиться с задачами, получить мудрый совет и поддержку от сообщества; ваш вопрос, заданный в общем канале, смогут найти ваши однокурсники — и ответы, которые вы получили, помогут и им.

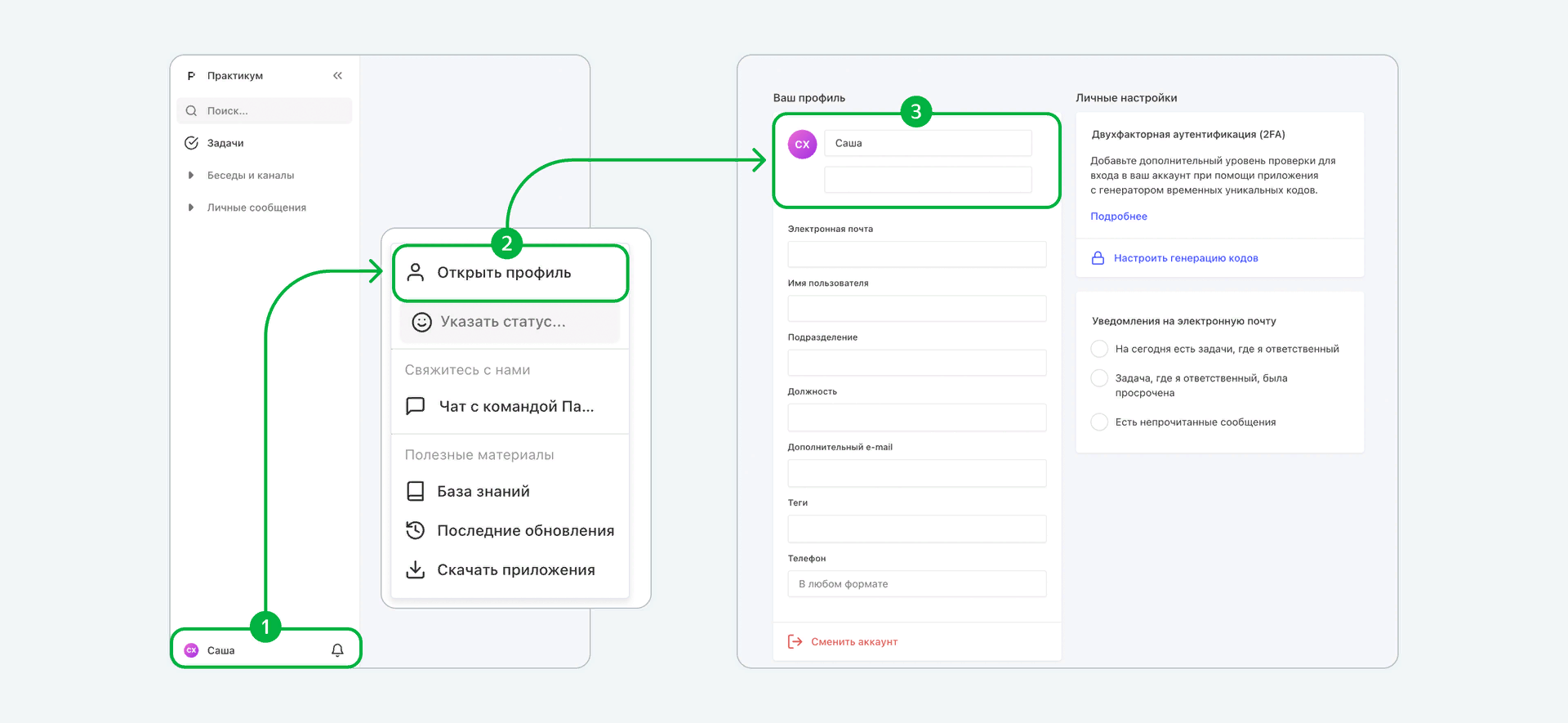
Мы составили короткую инструкцию, чтобы вам было проще освоиться в Пачке.

Внимание: не регистрируйтесь в мессенджере Пачка самостоятельно. Куратор пришлёт вам ссылку-приглашение, обязательно дождитесь этого момента. В ином случае вы не сможете авторизоваться в Пачке вашей когорты под своим адресом электронной почты.

## Оформление профиля

Чтобы вас можно было легко найти в мессенджере — заполните свой профиль:

1. Поставьте аватарку с вашей фотографией.
2. Укажите ваши имя и фамилию по-русски.



## Треды

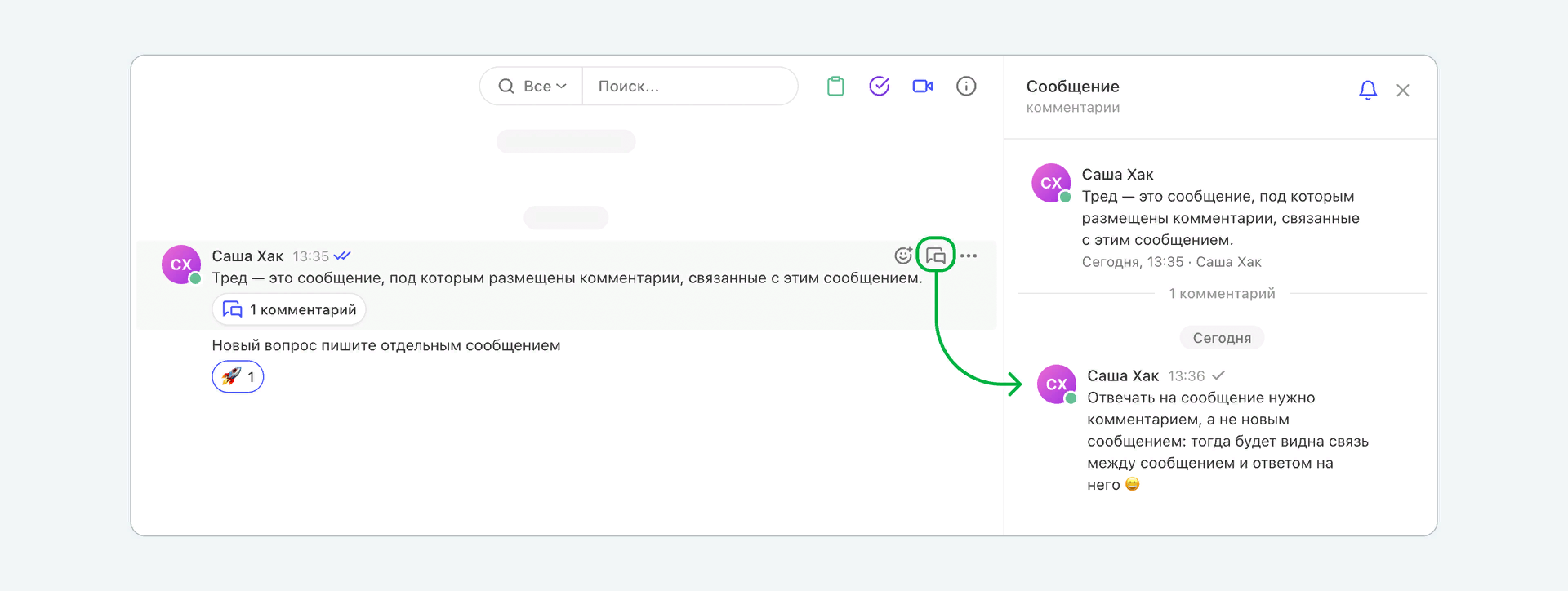
Сообщения в Пачке группируются по системе **тредов.** Тред — это сообщение, под которым размещены комментарии, связанные с этим сообщением.

Отвечать на сообщение нужно комментарием, а не новым сообщением: тогда будет видна связь между сообщением и ответом на него. Любое сообщение можно отредактировать, но нельзя удалить.

Основное правило:

* новый вопрос пишите отдельным сообщением,
* ответ на сообщение пишите в комментариях.

Тред можно откомментировать, сохранить или переслать; можно отметить сообщение смайликом.

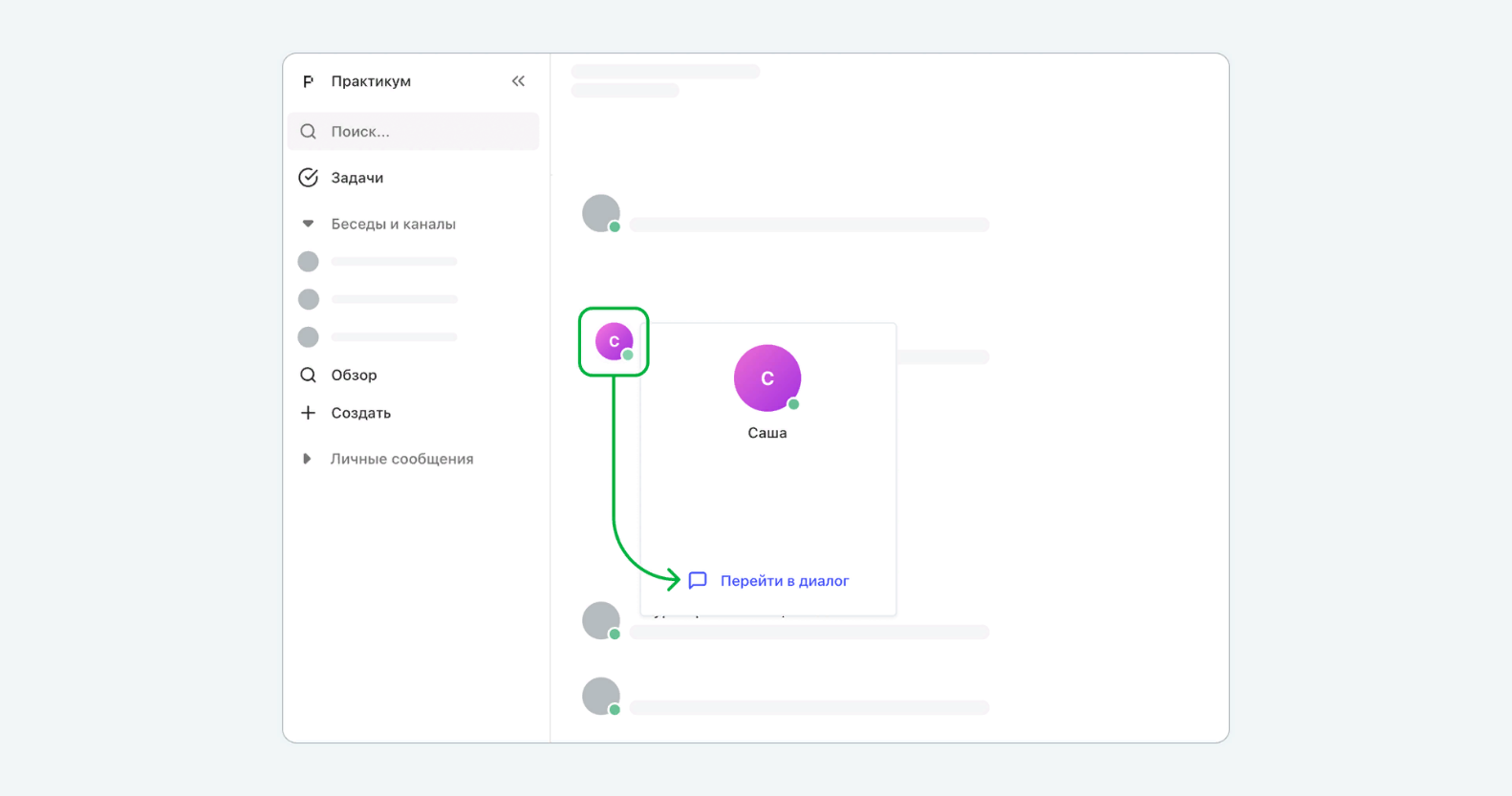


В тред можно призвать любого пользователя: поставьте символ @, начните писать его имя — и выберите его имя из всплывшей подсказки. После отправки такого сообщения пользователь, которого упомянули, получит уведомление.

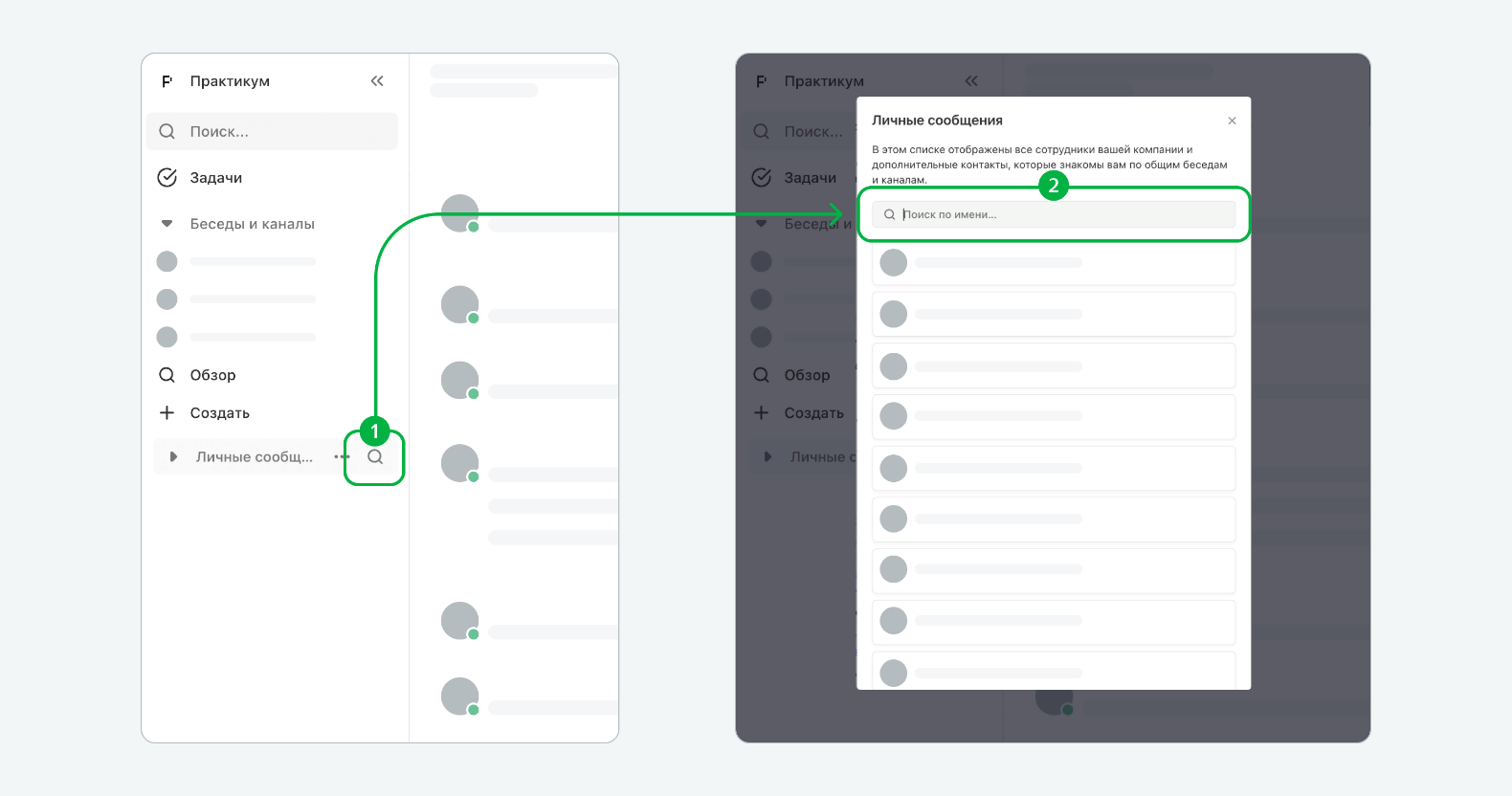
## Личные сообщения, диалоги

Пачка позволяет отправлять личные сообщения, которые увидит только получатель.

Чтобы открыть личный чат с участником сообщества, нажмите на имя пользователя и кнопку «Перейти в диалог»:



Есть и другой способ: в левой части экрана найдите раздел «личные сообщения», нажмите на лупу рядом с названием раздела и в поисковой строке введите имя человека, с которым хотите поговорить:



## Чего лучше не делать в личных сообщениях:

* Не стоит писать «Привет!», ждать, пока собеседник ответит — и лишь после его ответа излагать суть дела; лучше в одном сообщении и поздороваться, и задать вопрос либо изложить просьбу.
* В личных сообщениях лучше вести переписку в основной «ветке» чата, а не комментировать сообщения собеседника: в комментариях ответы могут затеряться.

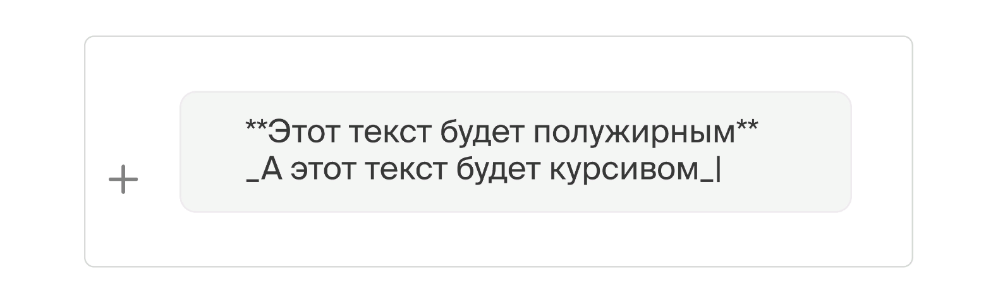
## Форматирование текста

* Начать новую строку: Shift+Enter (Windows) или Cmd+Enter (Mac)

Для изменения начертания в начале и в конце форматируемого текста ставятся специальные символы:

* выделить текст полужирным: обрамите текст двойными звёздочками;
* выделить текст курсивом: обрамите текст подчёркиваниями.

Набираем текст в окне сообщения:



## Как отправить код?

Код, отправленный в обычном текстовом сообщении, будет неудобно читать. В Пачке можно отправить форматированный фрагмент кода.

В новой строчке поставьте три знака апострофа: ```, затем вставьте код, и после кода — снова три апострофа.

Если кода много — можно воспользоваться сервисом <https://gist.github.com/>. Там вы можете сохранить свой код, скопировать уникальную ссылку и переслать её собеседнику в Пачке.

## Как отправить картинку?

Масса вариантов:

* перетащите картинку в Пачку, прямо в окошко, где вы пишете текст;
* вставьте картинку из буфера обмена, предварительно скопировав её откуда-нибудь: Ctrl+C → Ctrl+V (Windows) или Cmd+C → Cmd+V (Mac).
* выберите картинку на компьютере через Проводник/Finder, нажав на плюсик + слева от окна для ввода сообщения.

## Поиск в Пачке

Не тратьте время на прокручивание ленты в Пачке: ищите необходимую информацию через поисковую строку в левой верхней части экрана:

## Ваши беседы в Пачке

**#Info** — канал для организационных объявлений, важных анонсов, сообщений о технических работах/проблемах на платформе.

* Это канал куратора курса, здесь он публикует все важные объявления.
* **В этом канале вы не создаёте новых сообщений**: это делает только куратор.
* Если нужно уточнить детали какого-то сообщения — спрашивайте в комментариях под этим постом. Куратор увидит ваши вопросы и ответит.

**#theory** и **#project** — чаты для обсуждения теоретических и практических моментов в обучении; это такие же инструменты обучения, как уроки на платформе.

* Здесь вы задаёте вопросы наставнику по курсу, по заданиям и общие вопросы по проектам.
* Перед тем как задать вопрос, просмотрите — возможно, кто-то уже задал похожий вопрос и получил ответ.
* Формулируйте вопрос как можно более чётко, чтобы наставник правильно вас понял и быстро разобрался в проблеме.
* Если вы хотите прокомментировать уже существующие вопросы — пишите комментарий, отвечайте в треде.
* Все вопросы по учебному материалу задавайте наставнику только в общих каналах, а не в личных сообщениях: ради коммуникации в команде и для обогащения всех студентов такие вопросы обсуждаются только в общих чатах.

**#library** — чат, где можно делиться ссылками, полезными для практики и развития в профессии:

* писать могут все;
* можно отправлять только ссылки на полезные материалы;
* к каждой ссылке добавляйте описание: о чём это или почему это круто;
* если вам понравилась опубликованная кем-то ссылка или была полезна — отметьте это смайликом или комментарием в треде, автору будет приятно.

**#offtopic** — чат для неформального общения, различные темы, эмоции и переживания.

Единственное ограничение — никакого мата и взаимных оскорблений. Пожалуйста, обратите внимание, что этот канал кураторы и наставники смотрят в последнюю очередь. Не стоит публиковать в нём вопросы, на которые нужно получить оперативный ответ.

## Задание

Сразу после того, как вы получите от куратора ссылку-приглашение и оформите свой профиль в Пачке — зайдите в канал **#offtopic** и напишите пост-приветствие:

* В заголовке должно быть написано ваше имя, место, где вы живете, ваше основное занятие или фраза, которая вас хорошо описывает. В Пачке нет специального поля для заголовка поста; выделите текст заголовка отдельной строкой и отформатируйте его полужирным начертанием.
* В теле поста коротко расскажите, что для вас было самым сложным в первом спринте.
* Приложите к посту свою фотографию.
* В комментариях к посту дайте ссылку на ваш любимый фильм, сериал, или книгу.
* Не жалея смайликов и комментариев пройдитесь по аналогичным постам ваших одногруппников. Вам будет интересно, а им — приятно.

Пожалуйста, не пропустите это задание и обязательно попрактикуйтесь в новом для вас приложении. Когда вы освоите этот мессенджер и привыкнете к нему — у вас всегда будет возможность задать вопросы, получить профессиональную и эмоциональную поддержку или просто обсудить текущие дела: в чате всегда кто-нибудь не спит.

**Ожидание и реальность**

Одно из самых неприятных ощущений — осознать, что ожидания не оправдались. Чтобы этого не произошло во время курса, предлагаем сыграть в игру «Правда или ложь».

Мы покажем вам несколько утверждений об особенностях обучения на программе, а вы решите, что из этого правда, а что ложь.

Мы стремимся развить в вас самоходность, то есть умение двигаться «на собственной тяге». Как студенту — это качество поможет вам не пасовать перед трудностями во время обучения, как будущему профессионалу — постоянно расти.

# На этом инструктаж окончен

Итак, мы рассказали:

* какие принципы лежат в основе обучения;
* как устроена наша программа;
* кто и как будет поддерживать вас в процессе учебы;
* как пользоваться мессенджером и какие каналы коммуникации он открывает;
* как ставить цели, чтобы пройти курс полностью;
* что делать с непонятными местами и как относиться к своим ошибкам.

Если что-то из вышеперечисленного вам неясно — смело пройдите нужный урок снова.

Если всё понятно, то вы готовы приступить к обучению.

**Тема 2/4: О курсе**

# О курсе

Добро пожаловать на курс! Он посвящён работе с базами данных и с фреймворком Django.

## О чём пойдёт разговор

Для начала мы изучим несколько необходимых тем и инструментов, без которых бэкенд-разработчику не обойтись.

* **Объектно-ориентированное программирование**. Это методология, принцип языков программирования. ООП делает код компактным и удобным в разработке, в поддержке и развитии.
* **Локальное окружение.** В этом курсе тренажёром уже не обойтись, настало время превратить свой компьютер в рабочее место программиста. Установим софт, всё настроим.
* **Git.** Эта программа упрощает совместную работу нескольких разработчиков над кодом одного проекта. В этот раз в качестве коллег будут ваши код-ревьюеры.
* **Базы данных и SQL.** Эти технологии созданы для хранения данных и управления ими. Вы узнаете, как создавать базы и записывать в них данные, как запрашивать информацию из БД и как искать иголки в стогах полученных данных.
* **Фреймворк Django.** Веб-проект можно написать с чистого листа, а можно воспользоваться **фреймворком** — набором готовых инструментов для разработки и собрать сайт «из деталей», как из конструктора Лего. **Django** — один из распространённых и популярных фреймворков, написанных на Python.

Мы пройдём с экскурсией по этим темам, попрактикуемся в них — и после этого начнём работу над большим проектом.

## Что такое Django

Django — это такой завод, производящий веб-страницы. Некоторые станки уже стоят на своих местах, другие хранятся на складе, их можно подключать при необходимости. А часть «оборудования» разработчик проектирует и собирает самостоятельно.

The Web framework for perfectionists with deadlines

Это девиз Django, и в течение курса вы увидите, соответствует ли он действительности

Веб-фреймворк для перфекционистов с дедлайнами

Django был разработан в новостной редакции. Практика новостников требует часто и быстро запускать небольшие веб-проекты, посвящённые определённым темам, как запланированным (например, Олимпиаде), так и неожиданным (например, чрезвычайным происшествиям). Как правило, дедлайн при запуске таких проектов — «вчера». Разработчики газеты «Lawrence Journal-World» из Канзаса в 2003 году подготовили программную «заготовку», позволяющую в считанные часы настроить и запустить новый веб-проект.

В 2005 году фреймворк Django был опубликован, и с тех пор прошёл путь от россыпи файлов, упрощающих жизнь нескольким разработчикам, до популярного проекта, пережившего множество «болезней роста» и ставшего «взрослой» технологией.

Мы не будем создавать атмосферу редакции, гоняющейся за сенсациями, у нас не будет фотографий Человека-паука на передовице и не будет Кларка Кента среди журналистов. Мы займёмся изучением того, что вам потребуется для уверенной работы в команде программистов.

Конечно, за долгое время существования у Django появились свои тёмные углы, которые стоило бы почистить. Мы постараемся сделать так, чтобы не часто в них попадать.

Главное, что надо помнить — эта технология создана для того, чтобы упростить работу, сделать её хорошо и уложиться в сроки.

**Тема 3/4: Базы данных**

**1/17 Что вас ждёт в этом спринте**

Из этого спринта вы узнаете, как устроены базы данных, а также познакомитесь с SQL — языком, на котором пишут команды для управления реляционными базами данных.

Без практики в этой теме вы точно не останетесь: вы будете писать запросы на SQL, создавать таблицы в базе данных, сохранять в них информацию и извлекать её.

## Зачем мне это?

Наладить обмен информацией между базой данных и приложением— одна из востребованных задач, которую решают именно бэкенд-разработчики. Работа с базами данных — один из тех навыков, без которых бэкенд-разработчик неконкурентоспособен, а умение написать SQL-запрос вполне может пригодиться на собеседовании при устройстве на работу.

# 2/17 Введение в базы данных

Работа с данными — это большая часть работы с любым проектом. Информация о пользователях, описания и цены на товары и географические координаты и многое другое — всё это данные, которые необходимо сохранять, обрабатывать и предоставлять пользователям.

Для хранения информации придумали **базы данных** — системы для хранения информации и доступа к ней.

По способу хранения информации базы данных разделяют на две большие группы: реляционные и нереляционные.

### Реляционные базы данных

Считается, что первое проработанное описание идеи реляционных баз данных было предложено доктором Коддом из компании IBM в статье «Реляционная модель данных для больших, совместно используемых банков данных» в 1970 году. А первую коммерческую реляционную БД IBM выпустила в 1981-ом.

Реляционную базу данных можно представить в виде **набора таблиц с информацией**, хранящихся в одном пространстве — вроде excel-таблиц, хранящихся в одной папке.

Каждая строка таблицы в БД — это отдельная запись о каком-то объекте. В полях записи (в ячейках строки) могут храниться различные характеристики этого объекта. Каждый столбец таблицы должен иметь уникальное имя.

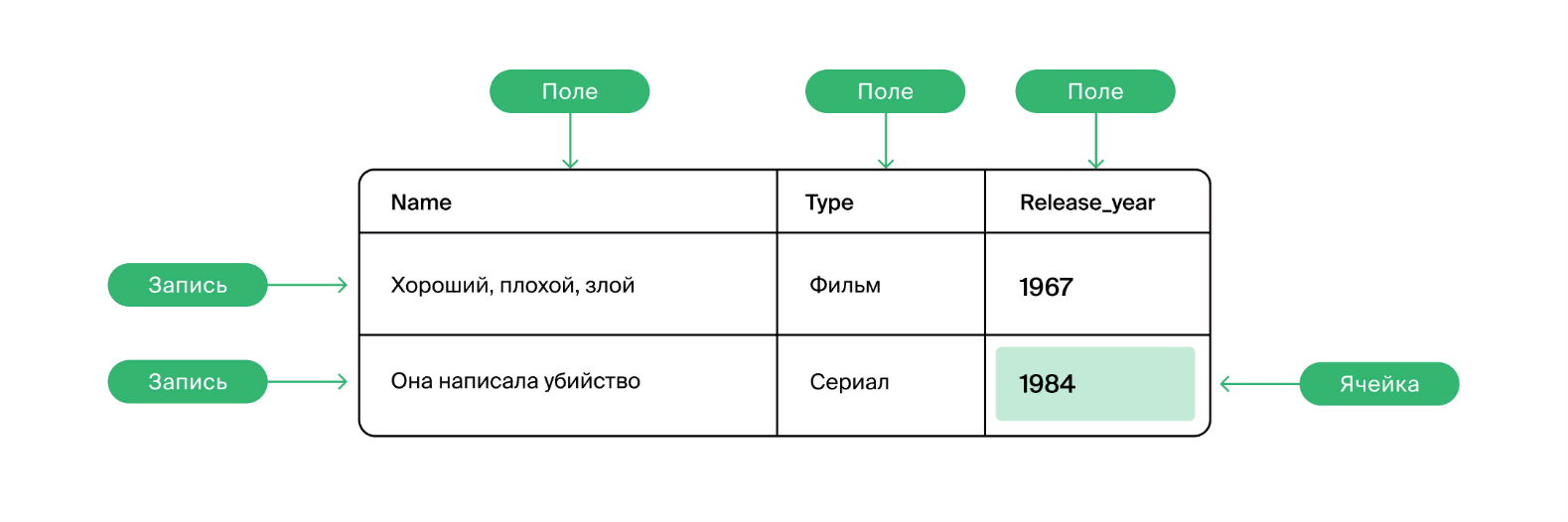
Таблица — это набор данных. Поэтому таблицам, как правило, дают названия во множественном числе, например — «заказы» (orders), «фильмы» (movies), «авторы» (authors). Но это — не обязательное правило.

Вот простая таблица **movies**, в ней собрана информация о нескольких фильмах. Столбцам принято давать имена в единственном числе, предельно короткие и содержательные, точно как переменным в Python.

| **name** | **type** | **release\_year** |
| --- | --- | --- |
| Весёлые Мелодии | Мультсериал | 1930 |
| Кто подставил кролика Роджера | Фильм | 1988 |
| Безумные Мелодии Луни Тюнз | Мультсериал | 1931 |
| Розовая пантера: Контроль за вредителями | Мультсериал | 1969 |

Элементы таблицы в реляционных базах данных принято называть так:

* **Запись (строка)** — это строка в таблице. Можно сказать «Таблица **movies** содержит четыре строки», а можно — «четыре записи».
* **Поле (столбец)** — это столбец таблицы или ячейка одной записи. Например: «В столбце type хранится тип фильма». Или так: «В поле release\_year хранится дата выхода фильма». Но можно сказать и так: «В записи о фильме «Кто подставил кролика Роджера» в поле release\_year хранится значение 1988».



### Типы данных в таблицах

В столбце таблицы реляционной БД хранятся данные одного типа; тип данных задаётся при создании таблицы. Например, колонка release\_year в приведённом примере хранит данные типа «число», и при попытке записать в это поле значение другого типа возникнет ошибка.

Самые востребованные типы полей:

* **Целое число**, как int в Python.
* **Число с плавающей точкой**, как float в Python.
* **Строка**, как str в Python (обычно применяется для коротких строк: названий, заголовков).
* **Булев тип:** логический тип данных. True или False.
* **Дата и время**. В базах данных обычно есть несколько специализированных типов для хранения дат, времени или временных интервалов.

Помимо базовых типов могут применяться и специальные, например — для хранения картографической информации, работы с деньгами, адресов компьютеров в сети или документов в формате JSON или XML.

Не во всех базах данных поддерживается полный набор типов данных. Например, SQLite поддерживает только пять из них:

* INTEGER — целое число;
* REAL — дробное число;
* TEXT — текст;
* BLOB — двоичные данные;
* NULL — специальное значение, означающее отсутствие информации.

Поэтому при работе с SQLite приходится идти на всякие хитрости:

* логические True и False хранятся в поле типа INTEGER в виде значений 0 и 1, но при получении информации проверка на истинность проводится с помощью идентификаторов TRUE или FALSE;
* при работе со временем и датой значения в базе данных сохраняют в виде строк или чисел.

## «Relation» значит «связь»

Реляционная база данных может содержать множество таблиц с различными данными; эти данные могут быть связаны между собой.

Вернёмся к таблице с фильмами: у каждого фильма есть режиссёр (а то и несколько), и о каждом режиссёре в БД можно хранить какую-то дополнительную информацию.

Для примера создадим таблицу, где будет имя режиссёра и год его рождения:

**directors**

| **id** | **name** | **birthday\_year** |
| --- | --- | --- |
| 1 | Текс Эйвери | 1908 |
| 2 | Роберт Земекис | 1952 |
| 3 | Джерри Чиникей | 1912 |

Каждый из этих режиссёров снимал какой-то фильм из таблицы **movies**, название фильма должно быть связано с его именем. Для этого в каждой записи таблицы **movies** можно указать, с каким режиссёром из таблицы **directors** связана эта запись.

Для организации связей между таблицами нужно твёрдо знать, какая запись на какую ссылается. Для точной «адресации» у каждой записи в каждой таблице должен быть уникальный идентификатор, **первичный ключ** (primary key, PK). Этот ключ должен быть уникален в пределах таблицы; в таблице **directors** PK режиссёров хранятся в колонке id.

В таблице, которая ссылается на другую таблицу, создают поле с атрибутом **внешний ключ** или **FK** (англ. foreign key); в этом поле указывается, на какой PK в другой таблице ссылается это поле.

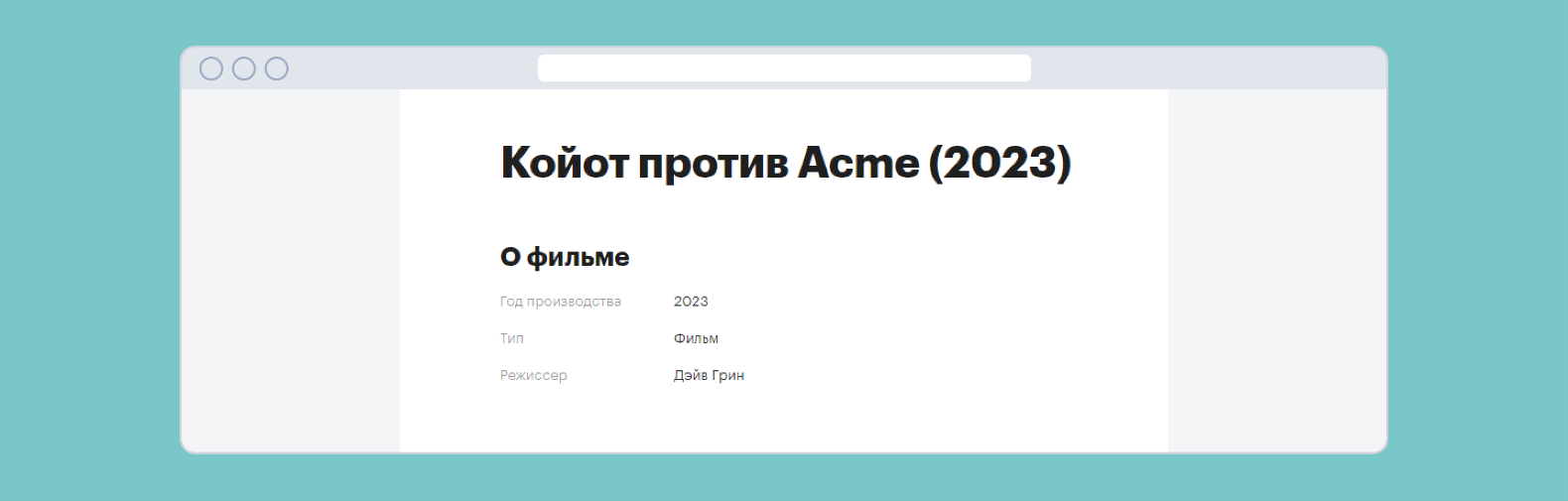
В таблице **movies** создадим дополнительную колонку director\_id — это будет **Foreign Key**, колонка, ссылающаяся на таблицу **directors**; в этой колонке в каждой строке укажем идентификатор того режиссёра, который снял фильм.

Заодно добавим в таблицу **movies** и колонку id, в ней будут храниться первичные ключи (уникальные идентификаторы) записей этой таблицы. Пригодятся.

**movies**

| **id** | **name** | **type** | **release\_year** | **director\_id** |
| --- | --- | --- | --- | --- |
| 1 | Весёлые Мелодии | Мультсериал | 1930 | 1 |
| 2 | Кто подставил кролика Роджера | Фильм | 1988 | 2 |
| 3 | Безумные Мелодии Луни Тюнз | Мультсериал | 1931 | 1 |
| 4 | Розовая пантера: Контроль за вредителями | Мультсериал | 1969 | 3 |

Теперь можно получить из таблицы movies запись о каком-нибудь фильме, и, взяв значение из поля director\_id, обратиться к нужной записи в таблице directors, получив из неё данные о режиссёре этого фильма. А потом вывести всю эту информацию на страницу сайта:



Связи могут быть гораздо сложнее описанного примера, но общий принцип остаётся неизменным.

В Django-проектах часто используются такие реляционные БД:

* [MySQL](https://www.mysql.com/).
* [PostgreSQL](https://www.postgresql.org/).
* [SQLite](https://www.sqlite.org/).

В этом курсе мы будем работать исключительно с реляционными базами данных. Начнём работу с SQLite, а позже поупражняемся с PostgreSQL.

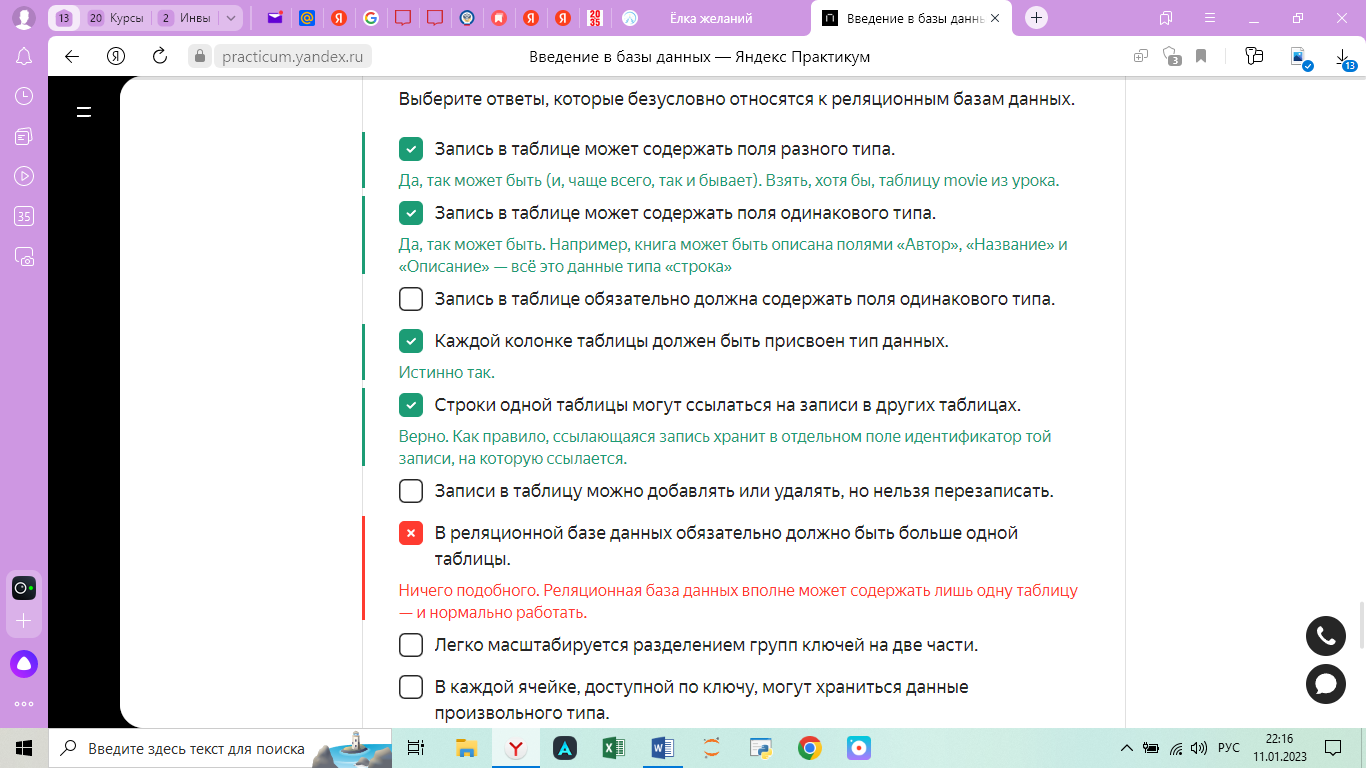
## Нереляционные базы данных (NoSQL)

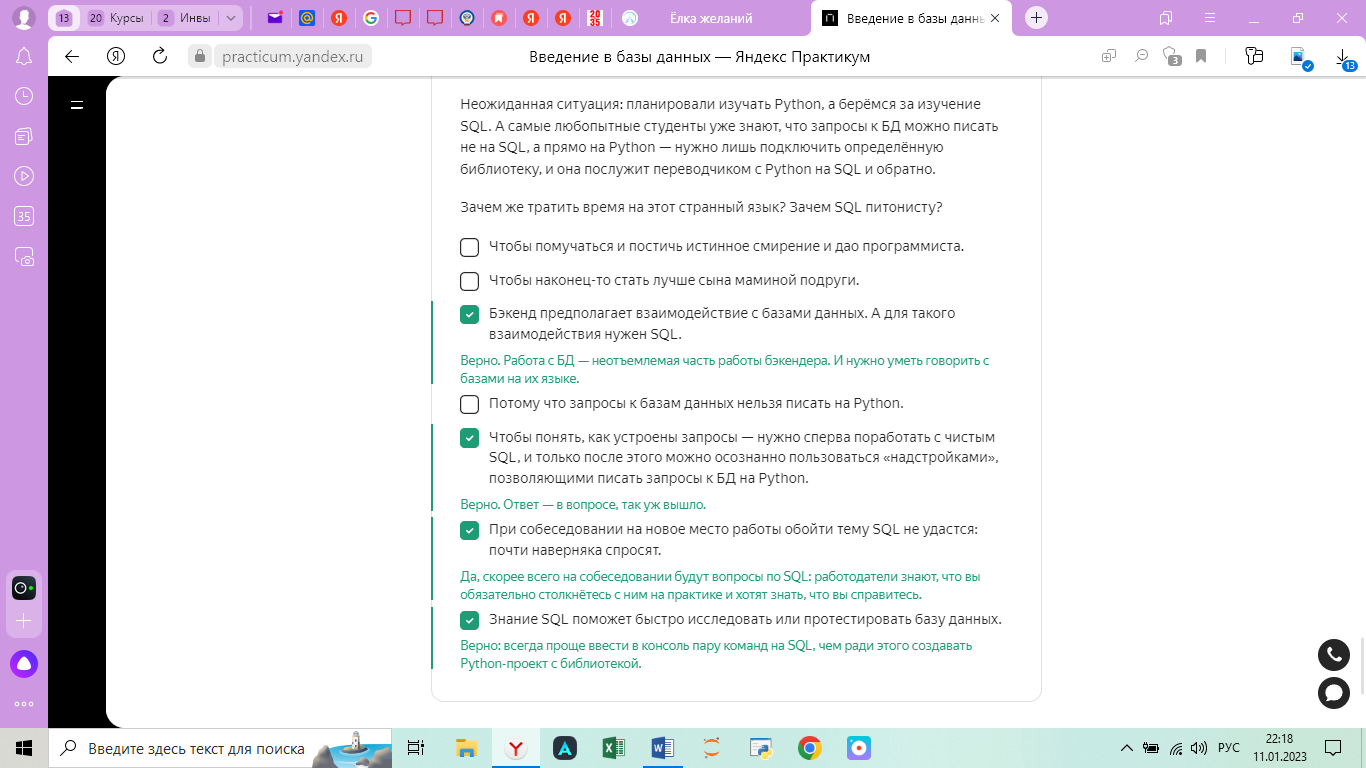
Реляционные БД требуют чёткой структуры. А сама информация, которую надо хранить может быть очень разнородной. Для работы с большими объёмами разнородной информации были придуманы **нереляционные базы данных** . Такие базы данных ещё называют **NoSQL (Not Only SQL)**.

Пионером в масштабном использовании нереляционных баз данных была компания Google. В начале 2000-х годов с их помощью решались проблемы масштабируемости и параллельной обработки данных в поисковой системе и в приложениях GMail, Maps, Earth.

Существует несколько типов NoSQL базы данных:

* **Ключ-значение** (Redis, Memcached, Amazon DynamoDB). Базу данных этого типа можно представить в виде таблицы, в каждой ячейке которой хранятся значения произвольного типа. **Главное преимущество таких баз — поиск по ключу и быстрое получение данных**. Обычно используются для кэшей объектов.
* **Документо-ориентированные** (MongoDB, CouchDB, Amazon DocumentDB). Похожи на БД типа ключ-значение, но единица хранения — «документ»; в документ можно записать любой набор данных. Такие базы часто используются для хранения информации о товарах с разнородными характеристиками.
* **Столбцовые (или колоночные)** (BigTable, HyperTable, Apache HBase, Apache Cassandra). В таких БД данные хранятся не в строчках, а в столбцах. Это позволяет выполнять быстрый поиск по базе. Область применения — большой объём данных, аналитика.
* **Графовые** (neo4j, AllegroGraph, ActiveRDF). Информация в таких БД представлена в виде узлов и отношений между узлами. Графовые БД позволяют эффективно работать с большими объёмами связанных данных; используются, например, для создания соцсетей.





# 3/17 СУБД и SQLite

База данных чем-то похожа на библиотеку: информация упорядочена, структурирована и разложена по полочкам. Этой информацией нужно как-то управлять.

Даже если посетитель библиотеки точно знает, какая книга ему нужна — требуется умение, чтобы найти запрошенную книгу среди тысяч томов.

А ведь запрос может быть и не точным: «Знаете, она была такая зелёная, на корешке надпись золотом и вензеля, про путешествие вокруг света по какой-то там параллели, ничего больше не помню». Эти задачи решает библиотекарь: он хорошо ориентируется в хранилище и знает, как найти нужную книгу. «Это же “Дети капитана Гранта” из “Библиотеки приключений”! Сейчас принесу.»

Базе данных тоже нужен такой управляющий: что толку хранить информацию, если ей нельзя воспользоваться.

## СУБД: система управления базами данных

Роль библиотекаря в базах данных выполняет **система управления базами данных**, **СУБД**.

Стоит дать ей команду, и она:

* создаст базу или таблицу в базе;
* внесёт новые данные или удалит ненужные;
* получит из базы нужную информацию по заданному условию;
* обеспечит безопасный доступ к данным.

## Очень простой SQL

Основной инструмент для работы с реляционными базами данных — специальный язык запросов: **Structured Query Language**, **SQL.**

Иногда аббревиатуру SQL в шутку расшифровывают как Simple Query Language — «простой язык запросов». Подвох в том, что SQL — это довольно сложный язык.

У SQL своеобразный синтаксис: в нём описываются условия **запросов**-команд: «база данных, сделай то-то и то-то с такими-то условиями».

Поработаем с базой данных, в которой есть таблица movies:

**movies**

| **id** | **name** | **type** | **release\_year** |
| --- | --- | --- | --- |
| 1 | Весёлые мелодии | Мультсериал | 1930 |
| 2 | Безумные Мелодии Луни Тюнз | Мультсериал | 1931 |
| 3 | Кто подставил кролика Роджера | Фильм | 1988 |
| 4 | Хороший, плохой, злой | Фильм | 1967 |
| 5 | Последний киногерой | Фильм | 1993 |
| 6 | Она написала убийство | Сериал | 1984 |
| 7 | Лас-Вегас | Сериал | 2003 |
| 8 | Паркер Льюис не проигрывает | Сериал | 1990 |
| 9 | Розовая пантера: Контроль за вредителями | Мультфильм | 1969 |
| 10 | Койот против Acme | Фильм | 2023 |

Отправляем **запрос**:

*-- Верни названия (значения поля name) тех фильмов из таблицы movies,*

*-- у которых в поле type записано «Фильм», а дата выпуска — ранее 1990 года*

*SELECT name FROM movies WHERE type = 'Фильм' AND release\_year < 1990;*

Ответ СУБД:

*-- Ваш заказ готов!*

*Кто подставил кролика Роджера*

*Хороший, плохой, злой*

*-- Кстати, в SQL тоже есть комментарии.*

*-- Ставьте двойной дефис в начале строки - и комментируйте на здоровье.*

В SQL можно оставлять **комментарии** двух типов: однострочные и многострочные. Однострочные начинаются с двух дефисов -- и продолжаются до конца строки. Многострочные комментарии начинаются со слеша и звёздочки /\* и заканчиваются ими же, в обратном порядке: \*/.

Язык SQL — это стандарт, который применяется в реляционных базах данных. В разных СУБД могут применяться дополнительные команды SQL, расширяющие стандартные возможности языка. Такие расширенные версии называют «диалектами» SQL.

В нашем курсе мы будем говорить на «классическом SQL» — применять команды, которые универсальны для всех диалектов SQL.

## Первые запросы, первые результаты

Выполним полный цикл операций: создадим и заполним таблицу в БД, а затем запросим содержимое этой таблицы. Всё как в жизни.

Для начала:

* создадим в пустой базе данных таблицу;
* добавим в эту таблицу пару записей

*CREATE TABLE movies(name TEXT, release\_year INTEGER);*

*-- СОЗДАТЬ ТАБЛИЦУ movies со столбцами(name TEXT, release\_year INTEGER);*

*-- Создаём запись в таблице movies:*

*INSERT INTO movies VALUES('Весёлые мелодии', 1930);*

*-- ВСТАВИТЬ В таблицу movies ЗНАЧЕНИЯ('Весёлые мелодии', 1930);*

*-- Создаём ещё одну запись в таблице movies:*

*INSERT INTO movies VALUES('Кто подставил кролика Роджера', 1988);*

*-- ВСТАВИТЬ В таблицу movies ЗНАЧЕНИЯ('Кто подставил кролика Роджера', 1988);*

В результате этих запросов

* в базе данных создана таблица movies с двумя столбцами — name и release\_year;
* в таблице созданы две записи.

Таблица выглядит так:

| **name** | **release\_year** |
| --- | --- |
| Весёлые мелодии | 1930 |
| Кто подставил кролика Роджера | 1988 |

Обратите внимание, что в SQL-запросах закрывающий символ ; обязателен. Команды SQL принято писать ЗАГЛАВНЫМИ буквами.

Язык запросов — регистронезависимый, то есть команда create table movies(name text, weight integer); тоже создаст таблицу. Но при таком стиле оформления плохо видно, где команды, а где сущности из базы, с которыми происходит работа.

А теперь из этой таблицы можно **получить** данные (например, для того, чтобы после обработки вывести их на страницу сайта или использовать как-то иначе):

*SELECT \* FROM movies;*

*-- ВЕРНУТЬ все поля (символ \* означает "все") записей ИЗ таблицы movies.*

В ответ на запрос вернутся данные, полученные из базы:

*Весёлые мелодии|1930*

*Кто подставил кролика Роджера|1988*

А можно сделать запрос и посложнее:

*SELECT name FROM movies WHERE release\_year = 1930;*

*-- ВЕРНУТЬ значения поля name ИЗ таблицы movies*

*-- тех записей, ГДЕ значение поля release\_year равно 1930.*

В ответе будет только содержимое поля name той записи, которая удовлетворяет условию WHERE:

*Весёлые мелодии*

# 4/17 Стиль оформления SQL-запросов

Два важных правила запомни для начала:

1. Операторы SQL пишутся в верхнем регистре: это позволяет визуально отделить их от ключевых слов.
2. Каждый оператор (а их в запросе может быть много) пишется с новой строки.

Вот SQL-запрос, написанный без форматирования:

*select catalog\_icecream.id,catalog\_icecream.is\_published,catalog\_icecream.is\_on\_main,catalog\_icecream.name,catalog\_icecream.category\_id,catalog\_icecream.text,catalog\_mainimage.id,catalog\_mainimage.is\_published,catalog\_mainimage.image,catalog\_mainimage.title,catalog\_mainimage.ice\_cream\_id from catalog\_icecream left outer join catalog\_mainimage on catalog\_icecream.id = catalog\_mainimage.ice\_cream\_id where catalog\_icecream.is\_on\_main and catalog\_icecream.is\_published;*

Этот запрос будет выполнен и вернёт запрошенную информацию, но прочесть такой код — сложная задача даже для квалифицированного аналитика.

Тот же запрос можно написать иначе:

*SELECT catalog\_icecream.id,*

*catalog\_icecream.is\_published,*

*catalog\_icecream.is\_on\_main,*

*catalog\_icecream.name,*

*catalog\_icecream.category\_id,*

*catalog\_icecream.text,*

*catalog\_mainimage.id,*

*catalog\_mainimage.is\_published,*

*catalog\_mainimage.image,*

*catalog\_mainimage.title,*

*catalog\_mainimage.ice\_cream\_id*

*FROM catalog\_icecream*

*LEFT OUTER JOIN catalog\_mainimage*

*ON catalog\_icecream.id = catalog\_mainimage.ice\_cream\_id*

*WHERE catalog\_icecream.is\_on\_main AND catalog\_icecream.is\_published;*

Читать стало проще, визуально понятно, где в запросе мухи, а где котлеты.

Стиль форматирования SQL-запросов в разных командах может отличаться, но общие принципы обычно сохраняются.

Оформить запрос по всем правилам можно вручную или с помощью SQL-форматера, например — [SQLFormat](https://sqlformat.org/).

Есть и [рекомендации, описывающие правила именования таблиц и столбцов](https://www.sqlstyle.guide/ru/#%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D1%8B) в них. Разумеется, это не догма, но знать их стоит.

Запрос то что надо: операторы написаны прописными буквами, переносы на месте:

*SELECT track\_id,*

*name,*

*composer,*

*milliseconds / 60000 AS minutes*

*FROM tracks*

*WHERE milliseconds > 350000;*

**5/17 Создание и наполнение информацией БД SQLite**

Работать с базой данных можно посредством «чистых» SQL-запросов, как это было в прошлых уроках, но есть другой подход: можно отправлять SQL-запросы из кода на Python.

В этом поможет библиотека **sqlite3**: с её помощью можно настроить соединение с БД и отправлять запросы. Устанавливать эту библиотеку не нужно: она встроена в Python и доступна для импорта.

Узнать подробности о работе с библиотекой **sqlite3** можно [в документации](https://docs.python.org/3/library/sqlite3.html).

Для начала нужно создать базу данных. Мы будем работать с SQLite, она хранит данные в файле; базу SQLite можно создать прямо из кода Python.

## Создание базы данных и таблиц

Вот как будет выглядеть код для создания таблиц movies и directors в базе данных с именем db:

*import sqlite3*

*# Если в текущей директории нет файла db.sqlite -*

*# он будет создан; сразу же будет создано и соединение с базой.*

*# Если файл существует, функция connect просто подключится к базе.*

*con = sqlite3.connect('db.sqlite')*

*# Создаём специальный объект cursor для работы с БД.*

*# Вся дальнейшая работа будет вестись через методы этого объекта.*

*cur = con.cursor()*

*# Готовим SQL-запросы.*

*# Для читаемости кода запрос обрамлён в тройные кавычки и разбит построчно.*

*cur.execute('''*

*CREATE TABLE IF NOT EXISTS directors(*

*id INTEGER PRIMARY KEY,*

*name TEXT,*

*birthday\_year INTEGER*

*);*

*''')*

*cur.execute('''*

*CREATE TABLE IF NOT EXISTS movies(*

*id INTEGER PRIMARY KEY,*

*name TEXT,*

*type TEXT,*

*release\_year INTEGER*

*);*

*''')*

*# Применяем запросы.*

*# Запросы, переданные в cur.execute(), не будут выполнены до тех пор,*

*# пока не вызван метод commit().*

*con.commit()*

*# Закрываем соединение с БД.*

*con.close()*

Метод execute() принимает одиночный SQL-запрос в виде обычной Python-строки. Нам нужно две таблицы — значит, вызываем метод execute() дважды.

Вот логика работы с базой данных:

1. Подключиться к базе данных, «создать соединение».
2. Подготовить SQL-запросы.
3. Применить запросы.
4. Закрыть соединение с БД.

Обратите внимание: запросы сначала подготавливаются, а после этого отправляются методом commit(); если не вызвать этот метод — подготовленные запросы не выполнятся.

**В базе данных не может быть двух таблиц с одинаковым названием. Поэтому перед созданием таблицы нужно сделать проверку IF NOT EXISTS. В результате новая таблица будет создана только в том случае, если таблицы с таким именем в базе нет.**

Если убрать из кода проверку IF NOT EXISTS, то в первый раз всё сработает, как надо — в базе появятся таблицы movies и directors; а вот при втором запуске программа упадёт с сообщением OperationalError: table movies already exists. До создания второй таблицы дело не дойдёт.

### Резидентные базы

Базу данных можно создать не в файле, а в памяти компьютера; такие базы называют **резидентными**.

Резидентная база создаётся так:

*con = sqlite3.connect(':memory:')*

После закрытия соединения резидентная база данных удаляется из памяти. Такие базы могут быть полезны для отладки и тестирования кода.

## Несколько SQL-запросов сразу

Для одновременного создания нескольких SQL-запросов в модуле sqlite3 есть метод executescript(). С его помощью запросы на создание двух таблиц можно записать компактнее:

*...*

*cur.executescript('''*

*CREATE TABLE IF NOT EXISTS directors(*

*id INTEGER PRIMARY KEY,*

*name TEXT,*

*birthday\_year INTEGER*

*);*

*CREATE TABLE IF NOT EXISTS movies(*

*id INTEGER PRIMARY KEY,*

*name TEXT,*

*type TEXT,*

*release\_year INTEGER*

*);*

*''')*

*...*

## Заполнение таблицы: INSERT

В базе готовы две таблицы, но в них пока нет ни одной записи.

Добавлять записи можно поштучно, вызывая метод execute() и передавая в него команду INSERT с нужными данными:

*...*

*cur.execute('''*

*INSERT INTO movies(id, name, type, release\_year)*

*VALUES (1, 'Весёлые мелодии', 'Мультсериал', 1930);*

*''')*

А можно передать в метод execute() значения полей в виде кортежа.

Для этого:

* в первом аргументе передаётся запрос с количеством заполняемых ячеек; ячейки обозначаются вопросительными знаками через запятую; например, четыре вопросительных знака — четыре ячейки;
* во втором аргументе — кортеж со значениями для полей. Модуль SQLite поймёт всё с полуслова и разместит значения в соответствующие ячейки.

*...*

*cur.execute(*

*'INSERT INTO movies VALUES(?, ?, ?, ?);',*

*(1, 'Весёлые мелодии', 'Мультсериал', 1930)*

*)*

*...*

При добавлении нескольких записей можно не повторять строку с вопросительными знаками. В sqlite3 есть shortcut executemany(), в него первым аргументом передаётся количество заполняемых ячеек, а вторым — список, каждый элемент которого — это кортеж, в котором хранятся значения полей.

Это позволяет быстро и просто создать несколько записей:

*...*

*directors = [*

*(1, 'Текс Эйвери', 1908),*

*(2, 'Роберт Земекис', 1952),*

*(3, 'Джерри Чиникей', 1912),*

*]*

*movies = [*

*(1, 'Весёлые мелодии', 'Мультсериал', 1930),*

*(2, 'Кто подставил кролика Роджера', 'Фильм', 1988),*

*(3, 'Безумные Мелодии Луни Тюнз', 'Мультсериал', 1931),*

*(4, 'Розовая пантера: Контроль за вредителями', 'Мультфильм', 1969),*

*]*

*cur.executemany('INSERT INTO directors VALUES(?, ?, ?);', directors)*

*cur.executemany('INSERT INTO movies VALUES(?, ?, ?, ?);', movies)*

*con.commit()*

*con.close()*

В результате выполнения этого кода три записи будут добавлены в таблицу directors и четыре — в таблицу movies.

1. Задача

Создаём и заполняем базу на раз-два-три. Задание первое: создайте базу данных SQLite. Файл должен называться db.sqlite.

Не забудьте импортировать модуль sqlite3.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

2. Задача

В базе данных *db.sqlite* создайте таблицу **ice\_cream** с полями name, description и category. Все поля должны хранить данные типа «строка».

В теории этого урока, в разделе «Создание базы данных и таблиц», есть всё необходимое; перечитайте его — и всё получится.

Для всех полей таблицы укажите тип данных: TEXT.

Не забудьте применить изменения con.commit() и закрыть соединение con.close()

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.execute('''*

*CREATE TABLE IF NOT EXISTS ice\_cream(*

*name TEXT,*

*description TEXT,*

*category TEXT*

*);*

*''')*

*con.commit()*

*con.close()*

3. Задача

Заполните таблицу ice\_cream данными из списка кортежей ice\_cream. Таблица с полями name, description и category уже создана.

Для заполнения информацией используйте executemany().

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*ice\_cream = [*

*('Пивное мороженое',*

*'Со вкусом светлого нефильтрованного лагера',*

*'Экзотическое',*

*),*

*('Мороженое с кузнечиками',*

*'В колумбийском стиле: с добавлением карамелизованных кузнечиков.',*

*'Экзотическое',*

*),*

*('Мороженое со вкусом сыра чеддер',*

*'Вкус настоящего сыра в вафельном стаканчике.',*

*'Экзотическое',*

*),*

*('Пломбир 1937',*

*'Пломбир по рецепту 1937 года Московского хладокомбината',*

*'Обычное'*

*),*

*]*

*cur.executemany('INSERT INTO ice\_cream VALUES(?, ?, ?);', ice\_cream)*

*con.commit()*

*con.close()*

**6/17 Получение данных: SELECT, FROM, WHERE, DISTINCT. Значение NULL**

Вы уже видели, как добавлять записи в таблицы; пришло время получить данные из БД. Для этого в SQL применяют конструкцию SELECT ... FROM ....

В этом уроке поработаем с новой базой данных: её содержимое будет похожим на прежнюю, но таблица в ней будет лишь одна.

| **id** | **name** | **type** | **release\_year** |
| --- | --- | --- | --- |
| 1 | Весёлые мелодии | Мультсериал | 1930 |
| 2 | Безумные Мелодии Луни Тюнз | Мультсериал | 1931 |
| 3 | Кто подставил кролика Роджера | Фильм | 1988 |
| 4 | Хороший, плохой, злой | Фильм | 1967 |
| 5 | Последний киногерой | Фильм | 1993 |
| 6 | Она написала убийство | Сериал | 1984 |
| 7 | Лас-Вегас | Сериал | 2003 |
| 8 | Паркер Льюис не проигрывает | Сериал | 1990 |
| 9 | Розовая пантера: Контроль за вредителями | Мультфильм | 1969 |
| 10 | Койот против Acme | Фильм | 2023 |

### Фильтрация по столбцам

После оператора SELECT указывают названия полей, значения которых должны вернуться в ответе; после FROM — названия таблиц, в которых надо искать данные.

*SELECT <перечень столбцов, значения из которых надо получить>*

*FROM <перечень таблиц, из которых надо получить данные>;*

В ответ на запрос SELECT СУБД возвращает данные в структурированном табличном виде. Чтобы не путаться в понятиях «таблица БД» и «таблица с ответом», возвращаемые данные будем называть **«результирующая выборка»** или просто **«выборка»**.

В ответ на запрос

*-- Вернуть только поля name, release\_year*

*SELECT name,*

*release\_year*

*-- из всех записей в таблице movies*

*FROM movies;*

…SQL вернёт такую результирующую выборку:

**

Python «под капотом» превращает полученную выборку в итерируемый объект, который можно перебрать в цикле; элементами этого объекта будут кортежи со значениями полей, полученных из БД.

Прочитаем и выведем в консоль все поля всех записей из таблицы **movies**:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*# Дорогой SQL, верни все столбцы всех записей из таблицы movies;*

*# символ \* после SELECT означает "верни все поля найденных записей".*

*cur.execute('''*

*SELECT \**

*FROM movies;*

*''')*

*# Python превращает результирующую выборку в итерируемый объект,*

*# который можно перебрать циклом:*

*for result in cur:*

*print(result)*

*# При получении данных из таблицы коммит не нужен.*

*con.close()*

В результате в консоли будут напечатаны кортежи:

*(1, 'Весёлые мелодии', 'Мультсериал', 1930)*

*(2, 'Безумные Мелодии Луни Тюнз', 'Мультсериал', 1931)*

*(3, 'Кто подставил кролика Роджера', 'Фильм', 1988)*

*(4, 'Хороший, плохой, злой', 'Фильм', 1967)*

*(5, 'Последний киногерой', 'Фильм', 1993)*

*(6, 'Она написала убийство', 'Сериал', 1984)*

*(7, 'Лас-Вегас', 'Сериал', 2003)*

*(8, 'Паркер Льюис не проигрывает', 'Сериал', 1990)*

*(9, 'Розовая пантера: Контроль за вредителями', 'Мультфильм', 1969)*

*(10, 'Койот против Acme', 'Фильм', 2023)*

Обычно при запросах к БД разработчику не нужны все поля записей. Например, на какой-то странице сайта нужно показать только названия и год выпуска для всех фильмов из базы. В такой ситуации нужно запросить из базы не все поля, а лишь два из них; для этого названия нужных столбцов перечисляют через запятую после SELECT.

Порядок полей важен, именно в этом порядке поля будут выгружены из таблицы.

*...*

*cur.execute('''*

*SELECT name,*

*release\_year*

*FROM movies;*

*''')*

*...*

Если теперь перебрать в цикле и распечатать содержимое объекта cur — снова будут напечатаны десять кортежей, но в каждом кортеже будет только два элемента: название фильма и год выпуска.

### Фильтрация по строкам

В запросе можно отфильтровать записи, удовлетворяющие заданным условиям. Найдём все фильмы, снятые после 1980 года.

Это можно сделать при помощи оператора WHERE.

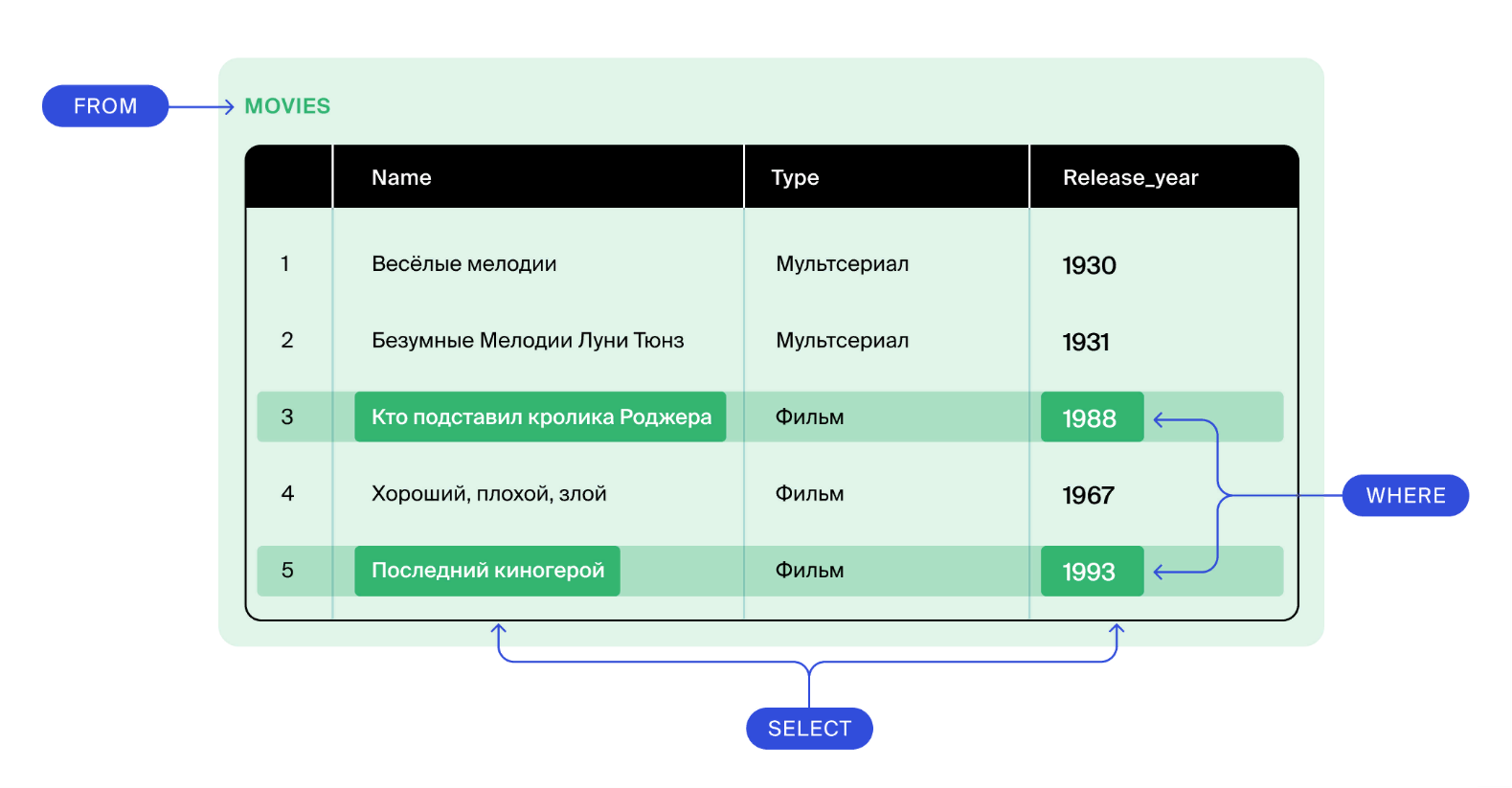
*SELECT name,*

*release\_year*

*FROM movies*

*-- ГДЕ значение поля release\_year больше 1980*

*WHERE release\_year > 1980;*

**

В результирующую выборку попадут только те записи, которые соответствуют условиям, описанным после оператора WHERE.

Для описания условий запроса применяют специальные операторы — большинство из них вам знакомы; вот несколько популярных:

* = — оператор **сравнения**, а не присваивания (это аналог == в Python).

*SELECT name*

*FROM movies*

*-- Найдём фильмы, выпущенные в 1988:*

*WHERE release\_year = 1988;*

* > , < — больше и меньше, >= , <= — больше или равно, меньше или равно.

*SELECT name*

*FROM movies*

*-- Найдём фильмы, выпущенные позже 1988:*

*WHERE release\_year > 1988;*

* <> — «не равно». В SQLite в качестве оператора неравенства можно применять и !=.

*SELECT name*

*FROM movies*

*-- Найдём фильмы, выпущенные не в 1988:*

*WHERE release\_year <> 1988;*

* BETWEEN ... AND ... — «между», для проверки значения в диапазоне: BETWEEN начало\_диапазона AND конец\_диапазона. Начало и конец диапазона включены в условие.

*SELECT name*

*FROM movies*

*-- Найдём фильмы, выпущенные с 1980 по 1990 год включительно:*

*WHERE release\_year BETWEEN 1980 AND 1990;*

* IN — вхождение в список:

*SELECT name*

*FROM movies*

*-- Найдём в базе все фильмы,*

*-- у которых значение поля type - 'Сериал' или 'Фильм':*

*WHERE type IN ('Сериал', 'Фильм');*

* LIKE — поиск строки по шаблону; в шаблонах можно применять символы-«маски»: знак процента (%) заменяет любой **набор символов**; символ подчёркивания \_ заменяет **один любой символ** (цифру, букву, пробел, пунктуационный или любой другой символ).

*SELECT name*

*FROM movies*

*WHERE type LIKE 'Мульт%';*

Этот запрос выберет все записи, где значение поля type — **Мульт**фильмы или **Мульт**сериалы. Символы \_ и % можно комбинировать в одном шаблоне.

При сравнении значения поля с числом — всё просто: WHERE название\_поля = 5.

Но если сравниваются строковые данные, то набор символов берут в одинарные кавычки: WHERE название\_поля = 'Сериал'.

Есть и другие операторы, их описание можно посмотреть в справочниках, например, [в этом](https://www.techonthenet.com/sqlite/comparison_operators.php).

Оператор WHERE позволяет использовать множество условий одновременно. Для объединения сразу нескольких условий используются операторы AND, OR или NOT, они работают так же, как в Python.

*...*

*cur.execute('''*

*SELECT name,*

*release\_year*

*FROM movies*

*WHERE (release\_year BETWEEN 1965 AND 1990) AND type LIKE '%ильм';*

*''')*

*for result in cur:*

*print(result)*

*...*

Если выполнить этот код — получится такой результат:

*('Кто подставил кролика Роджера', 1988)*

*('Хороший, плохой, злой', 1967)*

*('Розовая пантера: Контроль за вредителями', 1969)*

Когда используется несколько условий, каждое из них проверяется на истинность, и на выходе получается либо значение TRUE, если утверждение истинно, либо FALSE, если утверждение ложно. Затем каждый из логических операторов действует по своим правилам:

* AND возвращает TRUE, только если оба логических значения тоже TRUE.
* OR возвращает TRUE, если хотя бы одно логическое значение тоже TRUE.
* NOT меняет значение выражения на противоположное: TRUE на FALSE, а FALSE на TRUE.

### Специальное значение NULL

Значение NULL означает отсутствие данных. Основная особенность NULL в том, что это значение не равно ничему; любая попытка сравнить NULL с чем угодно, даже с NULL, вернёт False.

Для работы со специальными значениями используют отдельные операторы — IS NULL и IS NOT NULL. Оператор IS NULL при встрече с NULL вернёт True, а оператор IS NOT NULL наоборот: вернёт True, если значение не равно NULL.

Операторы IS NULL и IS NOT NULL можно применять в условиях, чтобы фильтровать данные.

Добавим в таблицу **movies** колонку gross: в ней будем хранить сумму кассовых сборов каждого фильма. Но какие-то фильмы не собрали вообще ничего (или мы просто не знаем о сборах) — и у них в поле gross стоит значение NULL.

Какие строки попадут в выборку в результате этого запроса к таблице **movies:**

SELECT \*

FROM movies

WHERE gross = NULL;



Все строки. Условие gross = NULL будет проигнорировано.



Тоже правильный ответ

Выборка будет пустой.



Неправильный ответ

Строки, у которых в поле gross стоит NULL.

Нет. Значение NULL не равно ничему, даже самому себе.

### DISTINCT

Этот оператор используется, чтобы получить уникальные значения из заданного столбца.

Выберем уникальные значения из колонки type в таблице movies:

*SELECT DISTINCT type*

*FROM movies;*

Получим перечень типов фильмов, без повторов:

Скопировать кодBASH

('Мультсериал',)

('Фильм',)

('Сериал',)

('Мультфильм',)

### Порядок операторов

В SQL-запросах важно указывать операторы в правильном порядке. Логика работы такая:

1. Выбрали столбцы из таблицы SELECT FROM.
2. В них выбрали значения, которые соответствуют условиям WHERE.

SELECT <перечень столбцов через запятую>

FROM <название таблицы>

WHERE <условия>;

SQL строг, и неправильный порядок операторов вызовет ошибку.

В SQL много ключевых слов, но SELECT, FROM и WHERE присутствуют практически в каждом запросе на получение информации.

**7/17 Практика запросов**

Для решения заданий вам понадобятся операторы

SELECT список\_столбцов\_которые\_нужно\_получить

FROM имя\_таблицы

WHERE условие\_по\_которому\_выбираются\_записи

А для решения первой задачи придётся применить методы дедукции.

1. Задача

В тренажёре заготовлена база данных db.sqlite. Выведите из таблицы, которая хранится в этой базе, значения столбцов name и text из всех записей.

Задача выполняется в два шага:

1. Узнать название таблицы.
2. Вывести записи из этой таблицы.

В уроках не было примеров запроса для получения списка таблиц в БД; найдите решение самостоятельно — поисковые системы вам помогут.

В SQLite есть служебная таблица sqlite\_master, в колонке tbl\_name она хранит названия всех таблиц базы, а в колонке type - тип объектов базы (таблицы, индексы и др.). Сделайте запрос к этой таблице и получите все значения поля tbl\_name, отфильтровав строки по полю type.

Код:

*cur = con.cursor()*

*# Напишите SQL запрос в строке.*

*cur.execute('''*

*SELECT tbl\_name*

*FROM sqlite\_master*

*WHERE type = 'table'*

*''')*

*table = cur.fetchone()[0] # Получите имя таблицы через атрибут курсора.*

*# Напишите SQL запрос в строке.*

*cur.execute(f'''*

*SELECT name,*

*text*

*FROM {table};*

*''')*

*for result in cur:*

*print(result)*

*con.commit()*

*con.close()*

Результат

('Классический пломбир', 'Настоящее мороженое, для истинных ценителей вкуса. Если на столе появляется пломбир — это не надолго.')

('Мороженое с кузнечиками', 'В колумбийском стиле: с добавлением карамелизованных кузнечиков.')

('Мороженое со вкусом сыра чеддер', 'Вкус настоящего сыра в вафельном стаканчике.')

('Пломбир 1937', 'Пломбир по рецепту 1937 года Московского хладокомбината.')

('Томатное мороженое', 'В СССР делали и томатное мороженое тоже. Вкус специфический.')

2. Задача

Из таблицы ice\_cream базы данных db.sqlite получите значение поля name тех записей, у которых значения полей is\_published и is\_on\_main равны TRUE.

Отфильтруйте значения с помощью WHERE и логического AND.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite') cur = con.cursor()*

*# Напишите SQL запрос в строке.*

*cur.execute('''*

*SELECT name*

*FROM ice\_cream*

*WHERE is\_published = TRUE AND is\_on\_main = TRUE;*

*''')*

*for result in cur:*

*print(result)*

*con.commit()*

*con.close()*

Результат

('Пивное мороженое',)

('Томатное мороженое',)

# 8/17 Сортировка, ограничение и сдвиг выборки: ORDER BY, LIMIT, OFFSET

Зачастую требуется не просто получить весь набор найденных записей или значений полей, но и:

* отсортировать выборку по какому-то принципу;
* ограничить выборку по количеству строк.

Продолжим работать с базой данных **movies**, хранящей список фильмов:

| **id** | **name** | **type** | **release\_year** |
| --- | --- | --- | --- |
| 1 | Весёлые мелодии | Мультсериал | 1930 |
| 2 | Безумные Мелодии Луни Тюнз | Мультсериал | 1931 |
| 3 | Кто подставил кролика Роджера | Фильм | 1988 |
| 4 | Хороший, плохой, злой | Фильм | 1967 |
| 5 | Последний киногерой | Фильм | 1993 |
| 6 | Она написала убийство | Сериал | 1984 |
| 7 | Лас-Вегас | Сериал | 2003 |
| 8 | Паркер Льюис не проигрывает | Сериал | 1990 |
| 9 | Розовая пантера: Контроль за вредителями | Мультфильм | 1969 |
| 10 | Койот против Acme | Фильм | 2023 |

Запросим список фильмов, выпущенных после 1980 года:

*SELECT name,*

*release\_year*

*FROM movies*

*WHERE release\_year > 1980;*

Распечатаем результат запроса:

('Кто подставил кролика Роджера', 1988)

('Последний киногерой', 1993)

('Она написала убийство', 1984)

('Лас-Вегас', 2003)

('Паркер Льюис не проигрывает', 1990)

('Койот против Acme', 2023)

Данные получены, но пользоваться ими неудобно. Надо бы отсортировать выборку по дате выхода картины в свет.

## Сортировка данных: ORDER BY

Для сортировки результатов запроса в SQL есть оператор ORDER BY. Обратите внимание: оператор сортировки пишется **после** условия WHERE. Это логично — ведь сначала нужно выбрать данные, и только после этого выстраивать их по порядку.

*SELECT name,*

*release\_year*

*FROM movies*

*WHERE release\_year > 1980*

*ORDER BY release\_year;*

Стало намного удобнее:

('Она написала убийство', 1984)

('Кто подставил кролика Роджера', 1988)

('Паркер Льюис не проигрывает', 1990)

('Последний киногерой', 1993)

('Лас-Вегас', 2003)

('Койот против Acme', 2023)

### Направление сортировки

По умолчанию, команда ORDER BY выполняет сортировку по возрастанию. Чтобы управлять направлением сортировки вручную, после имени столбца указывается ключевое слово ASC (по возрастанию) или DESC (по убыванию).

Вывести фильмы в порядке «от новых к старым» можно так:

...

ORDER BY release\_year DESC;

А оба этих запроса — равнозначны:

...

ORDER BY release\_year ASC;

...

ORDER BY release\_year;

Результирующую выборку можно отсортировать и по текстовым полям, тогда результаты будут выведены в алфавитном порядке. Сортировка с ключевым словом DESC «развернёт» порядок, и записи будут выведены в порядке от **Я** до **А**.

При сортировке можно указать несколько полей, перечислив их через запятую. Сначала результаты отсортируются по первому полю, а потом — по второму.

Отсортируем выборку по полю type в обратном порядке (от Я к А), а фильмы с одинаковым типом отсортируем по алфавиту:

*SELECT type,*

*name*

*FROM movies*

*WHERE release\_year > 1980*

*ORDER BY type DESC, name;*

Сначала сортируется тип картины в обратном алфавитном порядке (от Я к А), а внутри type — алфавитная сортировка по названиям.

('Фильм', 'Койот против Acme')

('Фильм', 'Кто подставил кролика Роджера')

('Фильм', 'Последний киногерой')

('Сериал', 'Лас-Вегас')

('Сериал', 'Она написала убийство')

('Сериал', 'Паркер Льюис не проигрывает')

Сортировка очень часто применяется на практике. В любом современном интернет-магазине, на новостном сайте или в блоге всегда есть сортировка по цене, рейтингу или дате публикации материала.

### Ограничение и сдвиг выборки: LIMIT и OFFSET

В ответ на запрос может вернуться большая выборка; но зачастую нужно получить лишь несколько записей из неё. Самый простой пример — постраничное отображение информации на сайте; допустим, на каждую страницу нужно вывести по два фильма из нашей базы данных. При запросе нужно получить только две записи, а не полный их список.

Для ограничения количества строк, возвращаемых в ответ на SQL-запрос, применяют оператор LIMIT (англ. «ограничение»); и OFFSET (англ. «смещение») .

LIMIT <сколько строк нужно показать> OFFSET <а на сколько строк сдвинуть выборку>;

Ограничим количество возвращаемых записей; выводить будем по алфавиту:

*SELECT name,*

*type*

*FROM movies*

*ORDER BY name*

*LIMIT 2;*

*Результат*:

('Безумные Мелодии Луни Тюнз', 'Мультсериал')

('Весёлые мелодии', 'Мультсериал')

При постраничном размещении информации эти две записи можно выкладывать на первую страницу. На вторую страницу нужно вывести третью и четвёртую записи.

Если нужно вернуть определённое число записей, но начинать нужно не с первой записи в выборке — после LIMIT ставят оператор OFFSET и в нём указывают, сколько записей нужно пропустить.

Вернём две записи из нашей выборки, но две первые пропустим:

*SELECT type,*

*name*

*FROM movies*

*ORDER BY name*

*LIMIT 2 OFFSET 2;*

В результате вернулись третья и четвёртая записи из отсортированного списка фильмов:

('Фильм', 'Койот против Acme')

('Фильм', 'Кто подставил кролика Роджера')

В переводе с SQL на русский:

* возьми значения полей type и name из таблицы movies
* отсортируй результаты по полю name в алфавитном порядке,
* покажи две строки, но пропусти две и начни с третьей.

Использование ограничений LIMIT и OFFSET без явной сортировки ORDER BY не вызовет ошибки, но может привести к непредсказуемым результатам. Ограничивать выборку лучше после сортировки.

Порядок операторов важен. Нельзя сортировать значения и ограничивать выборку без того, что бы сначала не указать таблицу и столбцы, которые должны попасть в выборку.

Порядок должен быть таким:

* SELECT
* FROM
* WHERE
* ORDER BY
* LIMIT OFFSET

Если не держать перед глазами таблицу — решить такую задачу будет трудновато. При запросах без ORDER BY в результирующей выборке записи будут расположены в порядке их размещения в БД. Предвидеть этот порядок почти невозможно, поэтому сортировка — залог предсказуемости при получении выборки.

Задача

В тренажёре заготовлена база данных db.sqlite с таблицей ice\_cream. Получите из этой таблицы результирующую выборку, которая

* содержит значения столбцов name и text...
* …тех записей, у которых значение поля is\_published равно True,
* отсортирована по полю name в обратном алфавитном порядке,
* содержит две записи, начиная со второй.
* Отфильтруйте значения с помощью WHERE.
* Отсортируйте записи с помощью ORDER BY и DESC.
* Примените LIMIT и OFFSET для ограничения итоговой выборки.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*# Напишите SQL запрос в строке.*

*cur.execute('''*

*SELECT name, text*

*FROM ice\_cream*

*WHERE is\_published = TRUE*

*ORDER BY name DESC*

*LIMIT 2 OFFSET 1;*

*''')*

*for result in cur:*

*print(result)*

*con.commit()*

*con.close()*

Результат:

('Пломбир 1937', 'Пломбир по рецепту 1937 года Московского хладокомбината.')

('Пивное мороженое', 'Со вкусом светлого нефильтрованного лагера.')

# 9/17 Агрегирующие функции COUNT, MIN, MAX, AVG, SUM

Отсортированная и ограниченная выборка — это удобно, но при работе с базой данных всегда хочется большего. В корзине интернет-магазина нужно посчитать сумму всех товаров, а для аналитика в ресторане важно узнать, как менялось число заказов и динамика среднего чека.

Для всех этих достойных целей в SQL существуют **агрегирующие** или **агрегатные** (от англ. aggregate, «сгруппированный, совокупный») функции. Такие функции **выполняют вычисления** на наборе значений и **возвращают не набор данных, а одиночное значение** — результат вычислений.

В общем виде запрос с агрегирующией функцией выглядит так:

SELECT АГРЕГИРУЮЩАЯ\_ФУНКЦИЯ(поле)

FROM Таблица;

Добавим в таблицу с фильмами столбец gross: в нём записаны кассовые сборы фильмов в долларах (по данным Кинопоиска). Для некоторых картин информации о сборах нет, поэтому в графе стоит NULL — этим ключевым словом в SQL обозначают отсутствие значения (в Python есть эквивалент NULL — None).

| **id** | **name** | **type** | **release\_year** | **gross** |
| --- | --- | --- | --- | --- |
| 1 | Весёлые мелодии | Мультсериал | 1930 | NULL |
| 2 | Безумные Мелодии Луни Тюнз | Мультсериал | 1931 | NULL |
| 3 | Кто подставил кролика Роджера | Фильм | 1988 | 156 452 370 |
| 4 | Хороший, плохой, злой | Фильм | 1967 | 25 118 063 |
| 5 | Последний киногерой | Фильм | 1993 | 137 298 489 |
| 6 | Она написала убийство | Сериал | 1984 | NULL |
| 7 | Лас-Вегас | Сериал | 2003 | NULL |
| 8 | Паркер Льюис не проигрывает | Сериал | 1990 | NULL |
| 9 | Розовая пантера: Контроль за вредителями | Мультфильм | 1969 | NULL |
| 10 | Койот против Acme | Фильм | 2023 | NULL |

### COUNT

Агрегирующая функция **COUNT()** (англ. «подсчёт») возвращает количество строк в результирующей выборке.

Этот запрос переводится с SQL на русский как «СОСЧИТАЙ все строки в таблице **movies»:**

SELECT COUNT(\*)

FROM movies;

В результате выполнения этого запроса вернётся число 10.

Через COUNT можно подсчитать число записей в любой выборке, например, с условием WHERE.

### MIN и MAX

Эти агрегирующие функции предназначены для поиска максимального или минимального значения в заданном столбце.

SELECT MIN(gross)

FROM movies;

*-- Вернёт: 25118063*

SELECT MAX(gross)

FROM movies;

*-- Вернёт: 156452370*

Функции MIN и MAX игнорируют значения NULL. Чего нет — то не может быть подсчитано.

Найти минимальное или максимальное значение можно и без агрегирующих функций — для этого сортируем таблицу по заданному полю и берём первое значение. Но такой запрос будет работать намного дольше, чем агрегатные функции.

### AVG и SUM

Часто требуется получить средние значения и итоговую сумму по какому то столбцу. SQL может вернуть и такие данные: для этого есть агрегирующие функции AVG (от англ. average «среднее») и SUM (от англ. «сумма»). Эти функции тоже игнорируют значения NULL. Подсчитываемая выборка может быть ограничена с помощью WHERE.

**Примеры использования:**

SELECT AVG(gross)

FROM movies

WHERE release\_year > 1980;

*-- Вернёт: 146875429.5*

SELECT SUM(gross)

FROM movies

WHERE release\_year > 1980;

*-- Вернёт: 293750859*

В веб-разработке **SUM** часто используется для подсчёта итоговой суммы в заказе, а **AVG** — для работы со всевозможными рейтингами.

1. Задача

В тренажёре заготовлена база данных db.sqlite с таблицей ice\_cream с такими полями:

* is\_published, тип BOOL
* is\_on\_main, тип BOOL
* name, тип TEXT
* text, тип TEXT
* category, тип TEXT
* price, тип REAL

Найдите и напечатайте стоимость самого дешёвого мороженого. Цены хранятся в поле price таблицы ice\_cream.

Агрегирующая функция MIN поможет решить задачу. Выборку ограничивать не надо.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*# Напишите SQL запрос в строке.*

*results = cur.execute('''*

*SELECT MIN(price)*

*FROM ice\_cream;*

*''')*

*for result in results:*

*print(result)*

*con.close()*

Результат:

(0.1,)

2. Задача

Посчитайте в таблице ice\_cream базы db.sqlite среднюю цену тех сортов мороженого, у которых в поле category стоит значение «Экзотическое». Результат выведите в консоль.

Агрегирующая функция AVG поможет в этой работе. Ограничите выборку с помощью WHERE по названию «Экзотическое» в поле category.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*# Напишите SQL запрос в строке.*

*results = cur.execute('''*

*SELECT AVG(price)*

*FROM ice\_cream*

*WHERE category = 'Экзотическое';*

*''')*

*for result in results:*

*print(result)*

*con.close()*

Результат:

(594.0,)

3. Задача

В таблице ice\_cream базы данных db.sqlite

* сорта мороженного, которые должны быть видны на сайте (опубликованы), обозначаются значением TRUE в поле is\_published;
* сорта мороженного, которые должны быть выведены на главную страницу проекта, обозначаются значением TRUE в поле is\_on\_main.

Посчитайте суммарную стоимость **опубликованных на главной странице**  сортов мороженого — тех, у которых одновременно есть статусы «**опубликовано»** и **«показать на главной»;** результат выведите в консоль.

Агрегирующая функция SUM поможет посчитать общую стоимость.

Ограничите выборку с помощью WHERE по названию значениям True полей is\_on\_main и is\_published.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*# Напишите SQL запрос в строке.*

*results = cur.execute('''*

*SELECT SUM(price)*

*FROM ice\_cream*

*WHERE is\_published = TRUE*

*AND is\_on\_main = TRUE;*

*''')*

*for result in results:*

*print(result)*

*con.commit()*

*con.close()*

Результат:

(500.1,)

# 10/17 Группировка GROUP BY и фильтрация HAVING

Для некоторых задач нужно группировать результирующую выборку по одному или нескольким столбцам. Такие группировки помогают, например, понять, насколько средний чек зависит от пола и возраста посетителя или от времени посещения ресторана.

## Группировка данных: GROUP BY

В таблице **movies** нет чеков и заказов, но и из неё можно получить полезную информацию с помощью группировки.

| **id** | **name** | **type** | **release\_year** | **gross** |
| --- | --- | --- | --- | --- |
| 1 | Весёлые мелодии | Мультсериал | 1930 | NULL |
| 2 | Безумные Мелодии Луни Тюнз | Мультсериал | 1931 | NULL |
| 3 | Кто подставил кролика Роджера | Фильм | 1988 | 156 452 370 |
| 4 | Хороший, плохой, злой | Фильм | 1967 | 25 118 063 |
| 5 | Последний киногерой | Фильм | 1993 | 137 298 489 |
| 6 | Она написала убийство | Сериал | 1984 | NULL |
| 7 | Лас-Вегас | Сериал | 2003 | NULL |
| 8 | Паркер Льюис не проигрывает | Сериал | 1990 | NULL |
| 9 | Розовая пантера: Контроль за вредителями | Мультфильм | 1969 | NULL |
| 10 | Койот против Acme | Фильм | 2023 | NULL |

Самый простой вариант использования **GROUP BY** — найти «группы», объединённые одинаковым значением в заданной колонке:

*SELECT type*

*FROM movies*

*GROUP BY type;*

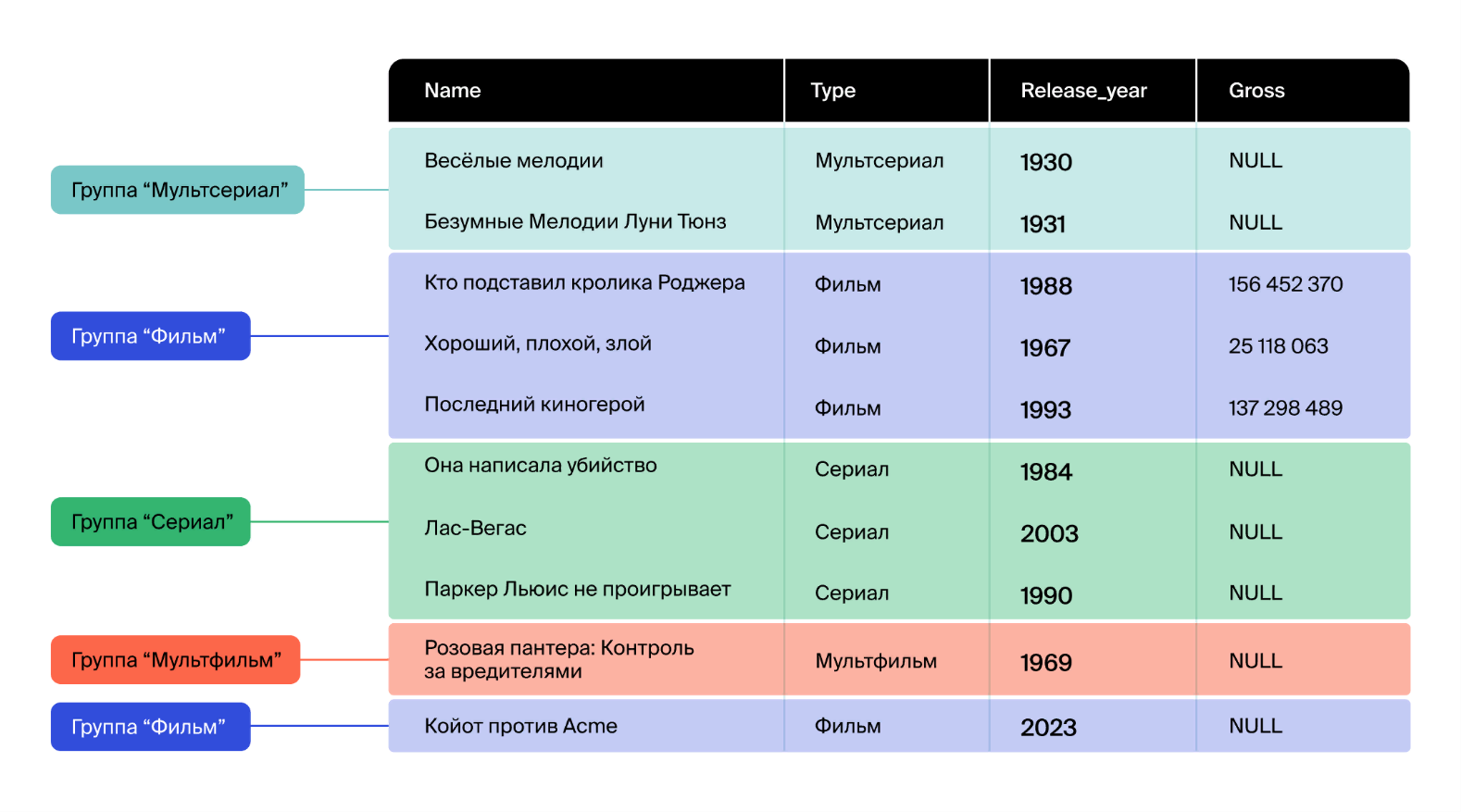
Все записи с одинаковыми значениями в поле type были объединены в группы, и в ответ на запрос вернулось по одной записи из каждой «группы»:

('Мультсериал',)

('Мультфильм',)

('Сериал',)

('Фильм',)



Аналогично ключевому слову **DISTINCT**, этот оператор может вернуть уникальные значения поля:

*SELECT type*

*FROM movies*

*GROUP BY type;*

*-- Тот же результат вернётся и при таком запросе:*

*SELECT DISTINCT type*

*FROM movies;*

Если после разбивки записей на группы к каждой из групп применить агрегирующие функции (**COUNT**, **MIN**, **MAX, AVG** или **SUM)**, будут доступны разные чудеса.

## Чудеса в примерах

**Посчитаем (COUNT), сколько в таблице фильмов каждого типа** — сколько фильмов, сколько сериалов, сколько мультфильмов и мультсериалов:

*SELECT type,*

*COUNT(\*)*

*FROM movies*

*GROUP BY type;*

Получилось!

('Мультсериал', 2)

('Мультфильм', 1)

('Сериал', 3)

('Фильм', 4)

**Найдём самый старый фильм (MIN)** в каждой группе:

*SELECT type,*

*MIN(release\_year)*

*FROM movies*

*GROUP BY type;*

Переведём запрос на русский: «в таблице movies собери в группы записи с одинаковым полем type; покажи значение type и минимальное значение release\_year для каждой из групп».

По результату этого запроса будет сформирована выборка:

('Мультсериал', 1930)

('Мультфильм', 1969)

('Сериал', 1984)

('Фильм', 1967)

**Просуммируем кассовые сборы (SUM)** для фильмов разного типа:

*SELECT type,*

*SUM(gross)*

*FROM movies*

*GROUP BY type;*

Вернётся такой ответ:

('Мультсериал', None)

('Мультфильм', None)

('Сериал', None)

('Фильм', 318868922)

В результате Python-магии NULL превратился в аналогичное по смыслу None.

Данные, которые попадут в группировку, можно предварительно отфильтровать через WHERE. Объединим фильмы в группы по типам и просуммируем кассовые сборы для каждой группы; но в группы включим лишь фильмы, выпущенные после 1990-го года; выведем название группы и сумму сборов — SUM(gross):

*SELECT type,*

*SUM(gross)*

*FROM movies*

*WHERE release\_year > 1990*

*GROUP BY type;*

Результат запроса:

('Сериал', None)

('Фильм', 137298489)

Заодно выяснили, что в нашей базе нет ни мультфильмов, ни мультсериалов младше 1990-го.

Следует иметь в виду, что для **GROUP BY** все значения NULL трактуются как равные, то есть при группировке по полю, содержащему **NULL**-значения, все строки с NULL попадут в одну группу.

Сгруппируем записи по одинаковым значениям в поле gross и выведем количество записей в каждой группе:

*SELECT gross,*

*COUNT(\*)*

*FROM movies*

*GROUP BY gross;*

И в результате:

*(None, 7)*

*(25118063, 1)*

*(137298489, 1)*

*(156452370, 1)*

## Фильтрация групп. HAVING

Оператор HAVING позволяет выполнить фильтрацию групп: он определяет, какие группы будут включены в результирующую выборку. Работа HAVING во многом аналогична применению WHERE; но WHERE применяется для фильтрации строк, а HAVING — для фильтрации групп.

Исключим из выборки все группы, в которых сумма кассовых сборов равна NULL:

*SELECT type,*

*SUM(gross)*

*FROM movies*

*GROUP BY type*

*-- Это условие относится к группам:*

*HAVING SUM(gross) IS NOT NULL;*

*-- ('Фильм', 318868922)*

Вот небольшая шпаргалка со списком самых востребованных операторов и ключевых слов, применяемых c получением данных из таблиц (SELECT) в SQL-запросах:

*SELECT ('столбцы (\* - для выбора всех столбцов); обязательно')*

*(могут применяться агрегатные функции COUNT, MIN, MAX, AVG и SUM; необязательно)*

*(и ключевое слово DISTINCT; необязательно)*

*FROM ('таблица; обязательно')*

*WHERE ('условие/фильтрация; необязательно')*

*GROUP BY ('столбец, по которому нужно сгруппировать данные; необязательно')*

*HAVING ('условие/фильтрация на уровне сгруппированных данных; необязательно')*

*ORDER BY ('столбец, по которому нужно ранжировать вывод; необязательно')*

*LIMIT ('сколько записей показывать; необязательно')*

*OFFSET ('сколько записей в выборке пропустить; необязательно')*

1. Задача

В тренажёре заготовлена база данных db.sqlite с таблицей ice\_cream. Цены хранятся в поле price.

Узнайте среднюю цену мороженого для каждой категории (по столбцу category) и выведите в консоль результат в виде (категория, средняя\_цена).

Агрегатная функция AVG поможет в решении. Выборку ограничивать не надо.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*# Напишите SQL запрос в строке.*

*cur.execute('''*

*SELECT category, AVG(price)*

*FROM ice\_cream*

*GROUP BY category;*

*''')*

*for result in cur:*

*print(result)*

*con.commit()*

*con.close()*

Результат:

('Обычное', 39.05)

('Экзотическое', 562.6666666666666)

# 11/17 Отношения между таблицами. ER-диаграммы

Попробуем превратить таблицу с перечнем фильмов в полноценную и гибкую базу данных; добавим информации и изменим архитектуру, создадим в базе несколько связанных таблиц. И на примере этой конструкции познакомимся с одной из основных идей реляционных баз — со связями между таблицами.

## Данные

Вот какая информация о фильме будет храниться в базе:

* Название (русское) — единственное.
* Оригинальное название — единственное
* Тип фильма — один из списка: мультфильм, сериал, мультсериал или фильм.
* Режиссёр — может быть один, а может быть и несколько.

## База данных

Создадим для этих данных таблицы в БД.

Кроме полезных данных в каждой таблице у записей будет уникальный идентификатор — колонка id.

### Таблицы

* Таблица **movies** (Фильмы). В ней будут храниться русские названия фильмов и ссылки на записи из других таблиц.
* Таблица **original\_names** (Оригинальные названия) — перечень оригинальных названий фильмов из таблицы **movies**.
* Таблица **types** (Типы) — перечень возможных типов фильмов (фильм, мультфильм и другие).
* Таблица **directors** (Режиссёры) — имена режиссёров.

К каждому фильму из таблицы **movies** должно быть привязано оригинальное название, тип и режиссёр.

Для привязки данных из других таблиц создадим в таблице **movies** поля, в которых будем указывать id нужной записи из таблиц **original\_names**, **types** и **directors.**

**movies**

| **id** | **name** | **original\_name\_id** | **type\_id** | **director\_id** |
| --- | --- | --- | --- | --- |
| 1 | Безумные Мелодии Луни Тюнз | 3 | 2 | 1 |
| 2 | Весёлые мелодии | 6 | 2 | 1 |
| 3 | Кто подставил кролика Роджера | 5 | 3 | 2 |
| 4 | Хороший, плохой, злой | 4 | 3 | 3 |
| 5 | Последний киногерой | 1 | 3 | 4 |
| 6 | Она написала убийство | 2 | 4 | 5, 6, 7 |

У последнего фильма три режиссёра; пока что отметим их так.

В таблице **movies** в полях original\_name\_id, type\_id и director\_id указаны id соответствующих записей в других таблицах.

Названия полей, ссылающихся на другие таблицы, могут быть любыми, но лучше составлять их по принципу имяТаблицы\_имяПоля.

В остальных таблицах будет храниться такая информация:

**original\_names**

| **id** | **name** |
| --- | --- |
| 1 | Last Action Hero |
| 2 | Murder, She Wrote |
| 3 | Looney Tunes |
| 4 | Il Buono, il brutto, il cattivo |
| 5 | Who Framed Roger Rabbit |
| 6 | Merrie Melodies |

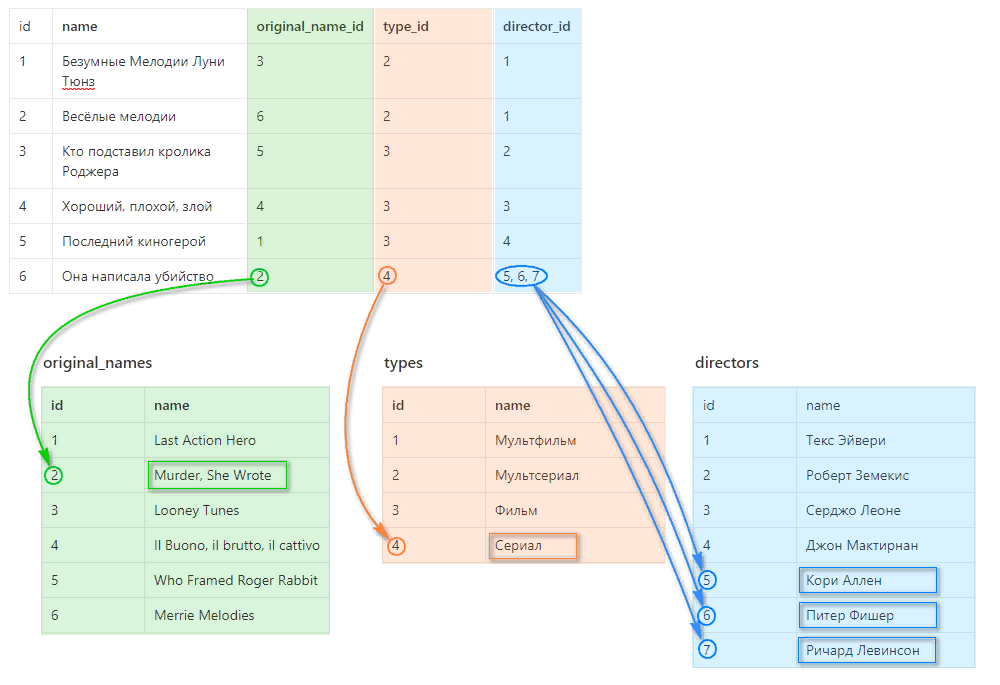
**types**

| **id** | **name** |
| --- | --- |
| 1 | Мультфильм |
| 2 | Мультсериал |
| 3 | Фильм |
| 4 | Сериал |

**directors**

| **id** | **name** |
| --- | --- |
| 1 | Текс Эйвери |
| 2 | Роберт Земекис |
| 3 | Серджо Леоне |
| 4 | Джон Мактирнан |
| 5 | Кори Аллен |
| 6 | Питер Фишер |
| 7 | Ричард Левинсон |

Можно схематично изобразить ссылки на другие таблицы для фильма «Она написала убийство»:



Зная связанные с записью идентификаторы из других таблиц, можно по цепочке извлечь дополнительные данные и получить полную информацию о фильме:

* **Русское название:** Она написала убийство
* **Оригинальное название:** Murder, She Wrote
* **Тип:** Сериал
* **Режиссёр:** Кори Аллен, Питер Фишер, Ричард Левинсон

Поля между собой связаны по-разному:

* У каждого фильма есть лишь одно оригинальное название, значит, два разных фильма не могут ссылаться на одну и ту же запись в таблице **original\_names.**
* Разные фильмы из таблицы **movies** могут ссылаться на одну и ту же запись в таблице **types.** Например, Безумные Мелодии Луни Тюнз и Весёлые мелодии — это два мультсериала.
* Есть ещё один вариант: у одного фильма может быть несколько режиссёров, то есть один фильм может ссылаться на несколько записей из таблицы **directors** (как, например, Она написала убийство); но и режиссёр может снять несколько фильмов (как Текс Эйвери в нашей базе).

Всё это — различные **типы связей**.

## Связи между таблицами

В реляционных базах данных может быть три типа связей:

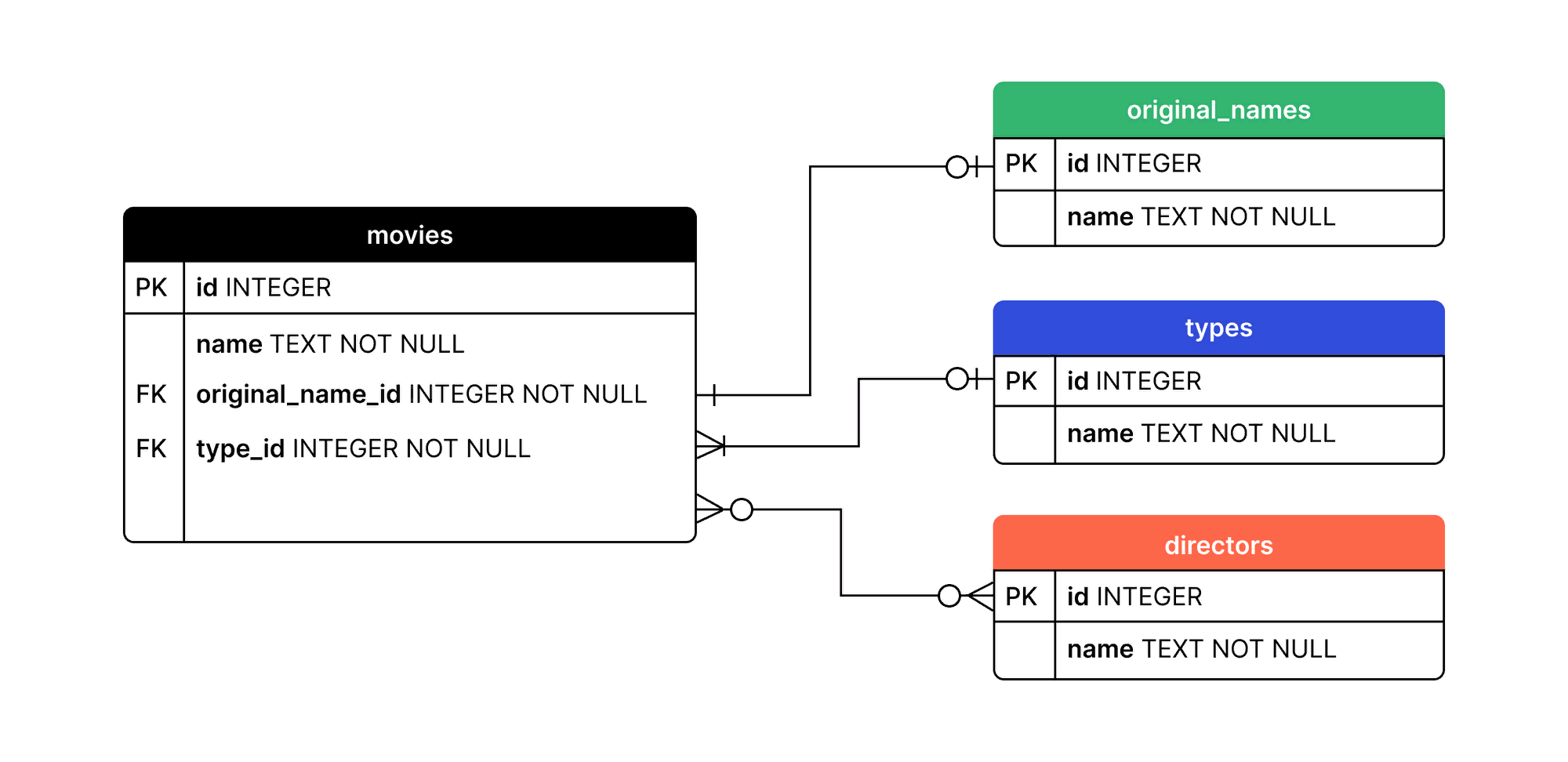
* Один-к-одному (1:1 или one-to-one): фильм → оригинальное название.
* Один-ко-многим (иногда отмечается 1:M или one-to-many): тип → фильмы.
* Многие-ко-многим (M:M или many-to-many): фильмы → режиссёры.

Чтобы настроить эти связи — в каждой из таблиц должен быть primary key (PK, значение, уникальное в рамках одной таблицы), а в таблицах, которые ссылаются на другие таблицы, понадобятся поля с внешними ключами (FK, foreign key), в которых будет указываться, куда мы ссылаемся.

### ER-диаграммы: схема базы данных

При проектировании БД таблицы и их связи изображают схематически; такие схемы называют **ER-диаграммы** (от англ. Entity-Relationship diagram, «диаграмма сущность — связь»). В схемах описываются таблицы, типы полей и связи с другими таблицами.

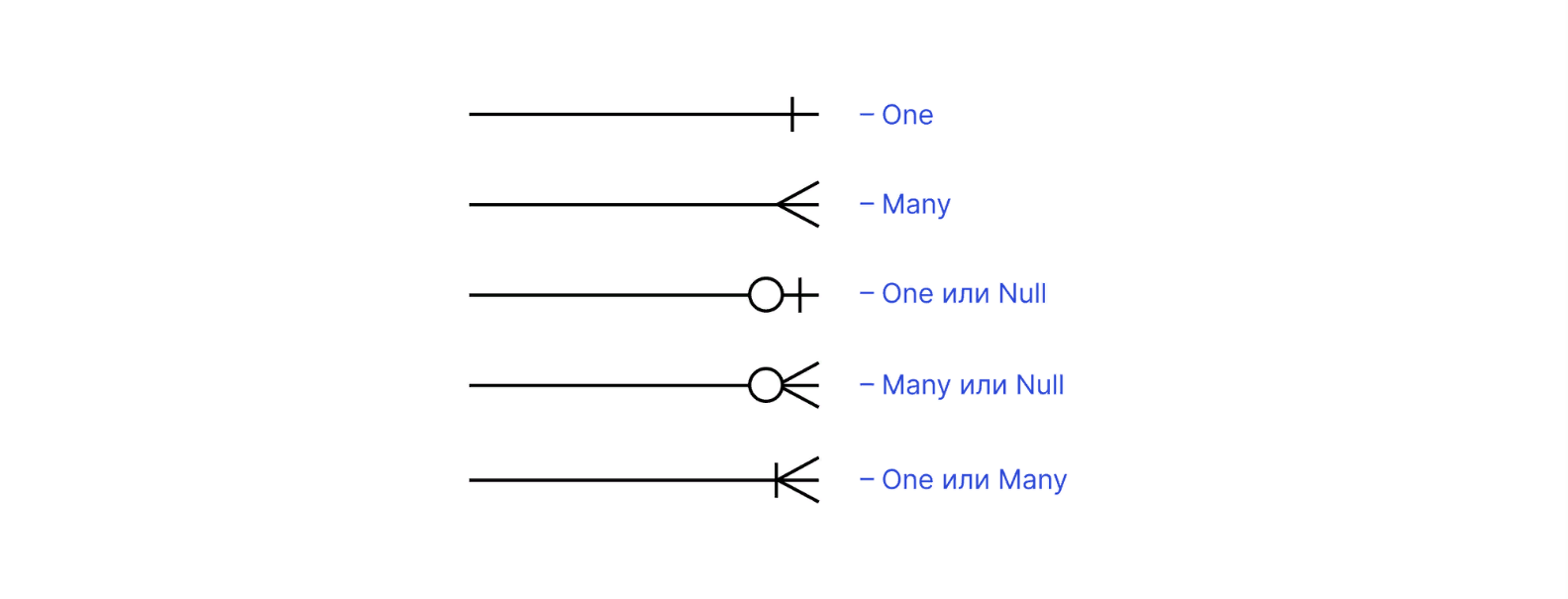
В упрощённом виде ER-диаграмма базы фильмов будет выглядеть примерно так:



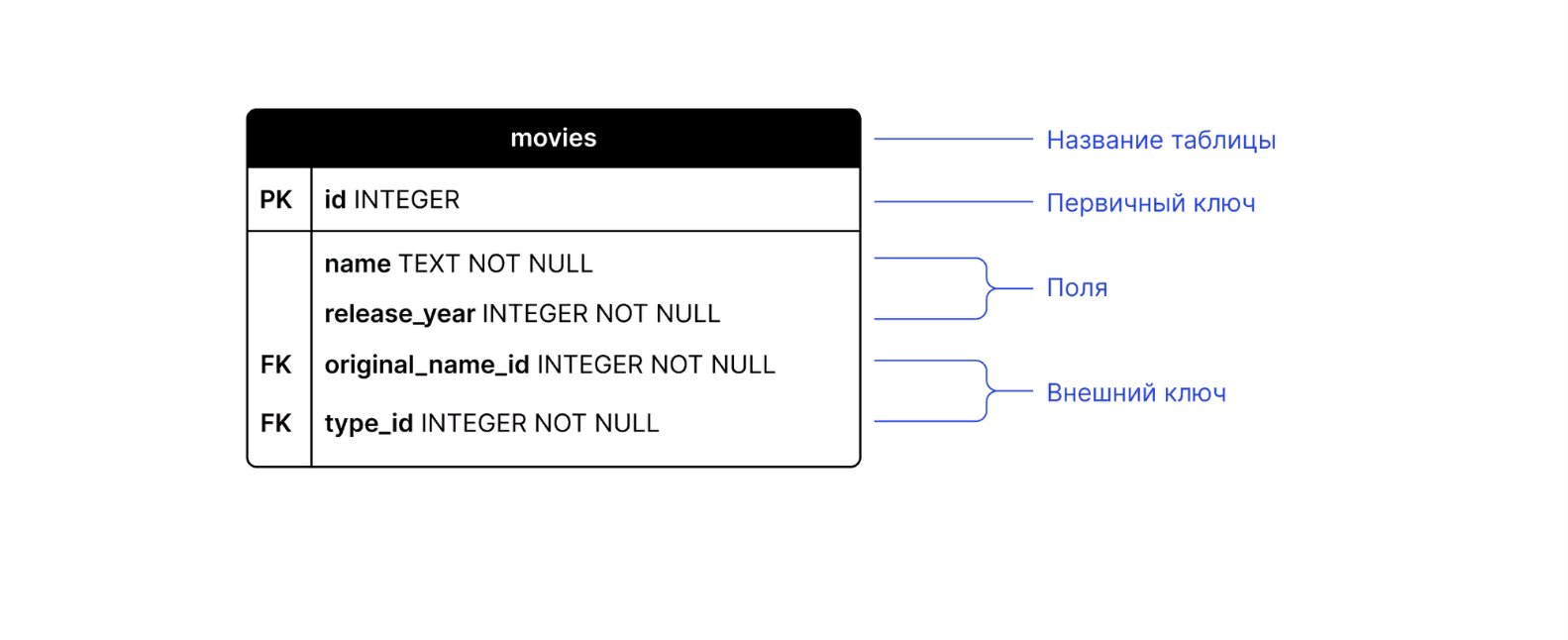
Обратите внимание, что связи идут от внешнего ключа (FK) к первичному (PK).

### Что ещё за значки на схеме?

Типы связей на ER-схемах обозначаются специальными символами:



Таким образом, связь между таблицами **movies** и **original\_names** — one-to-one, между таблицами **types** и **movies** — one-to-many, а между **movies** и **directors** — many-to-many.



## Связь «один-к-одному» (1:1)

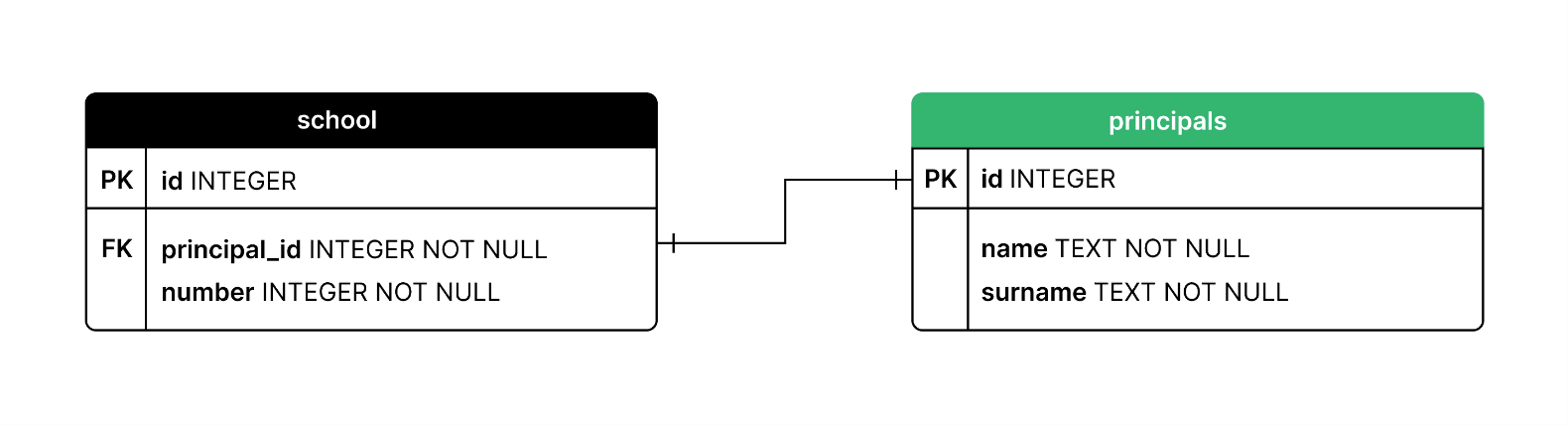
В жизни отношения один-к-одному довольно часты:

* **Школа — директор.** В одной школе — один директор.
* **Машина — VIN-код.** У одного автомобиля — один уникальный номер.
* **Человек — рисунок радужной оболочки глаза.** Рисунок уникален для каждого человека — одинаковых глаз у разных людей быть не может.

Но в базах данных связь один-к-одному применяют редко: проще объединить две таблицы в одну, чем использовать эту связь.

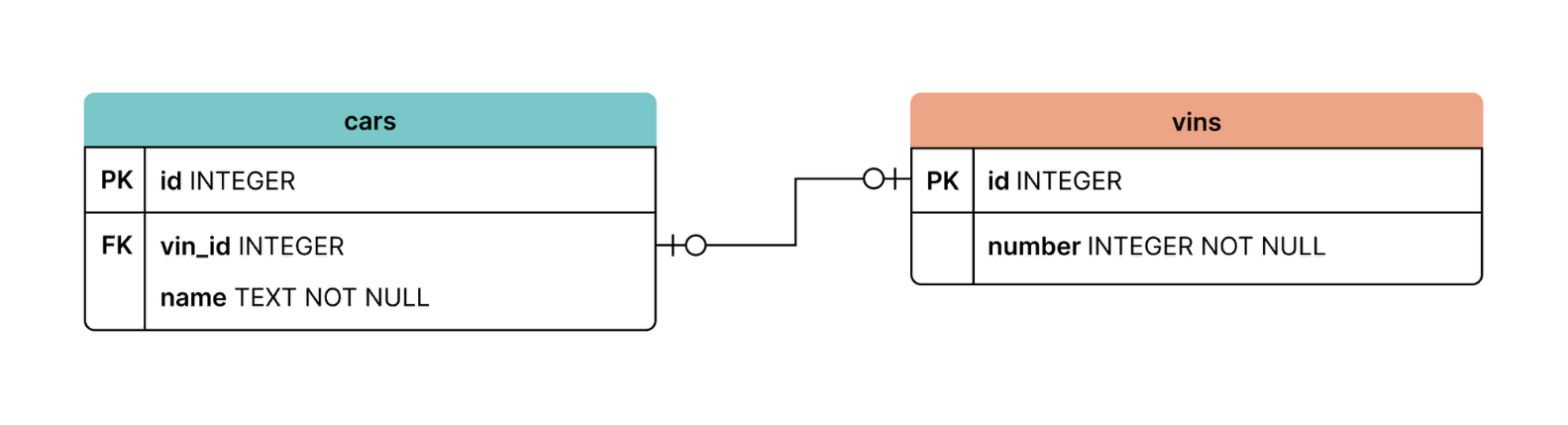
### Один-к-одному: обязательная связь

В школе обязательно есть директор (иначе кто же будет вызывать родителей?). Связь между таблицами **schools** и **principals** — обязательная, в поле principal\_id не может стоять NULL.



### Один-к-одному: необязательная связь

Связь между автомобилем и его VIN-кодом. У автомобиля Ford Model T, выпущенного в 1908-м году, никакого VIN-кода не было, а у современных машин он есть. Значит, в таблице **cars** ссылка на **vins** может стоять, а может и нет: значением поля vin\_id в таблице **cars** может быть NULL.



## Связь «один-ко-многим» (1:M)

Самая часто встречающая связь в реляционных БД — это отношение «один-ко-многим» и её брат-близнец «многие-к-одному». В зависимости от точки зрения эту связь можно описывать и так, и так.

Эта связь описывает, например, такие отношения:

* **Классы в школе**: в одной школе есть много классов (один-ко-многим). Но каждый класс принадлежит только одной школе (многие-к-одному).
* **Кинотеатры и экраны**: в одном кинотеатре может быть много экранов (один-ко-многим), но множество экранов принадлежит только одному кинотеатру (многие-к-одному).
* **Место рождения**: в городе может родиться много детей, но каждый из этих детей рождён в одном городе.
* **Стоянка и машины**: на одной стоянке может быть много машин, но каждая машина запаркована только на одной стоянке.

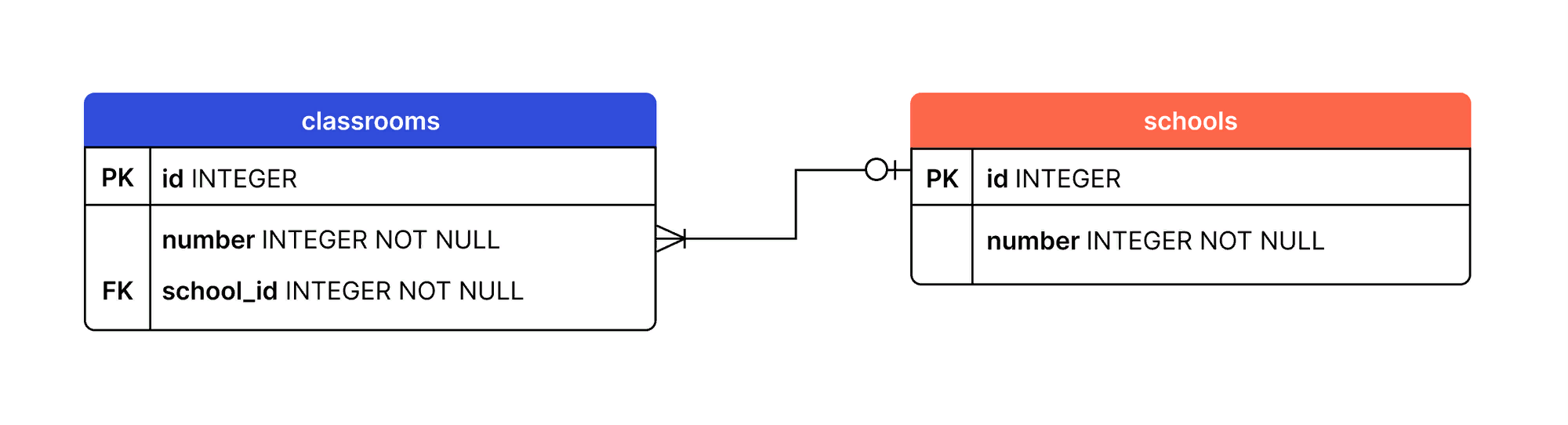
Основная идея связи 1:M в том, что с какой-то одной сущностью можно связать много других. Например, у фильма есть тип (мультфильм, фильм, сериал…); данные о типах хранятся в таблице **types**. К одному типу может принадлежать много разных фильмов.

Связи 1:M могут быть обязательные и нет.

### Один-ко-многим: обязательная связь

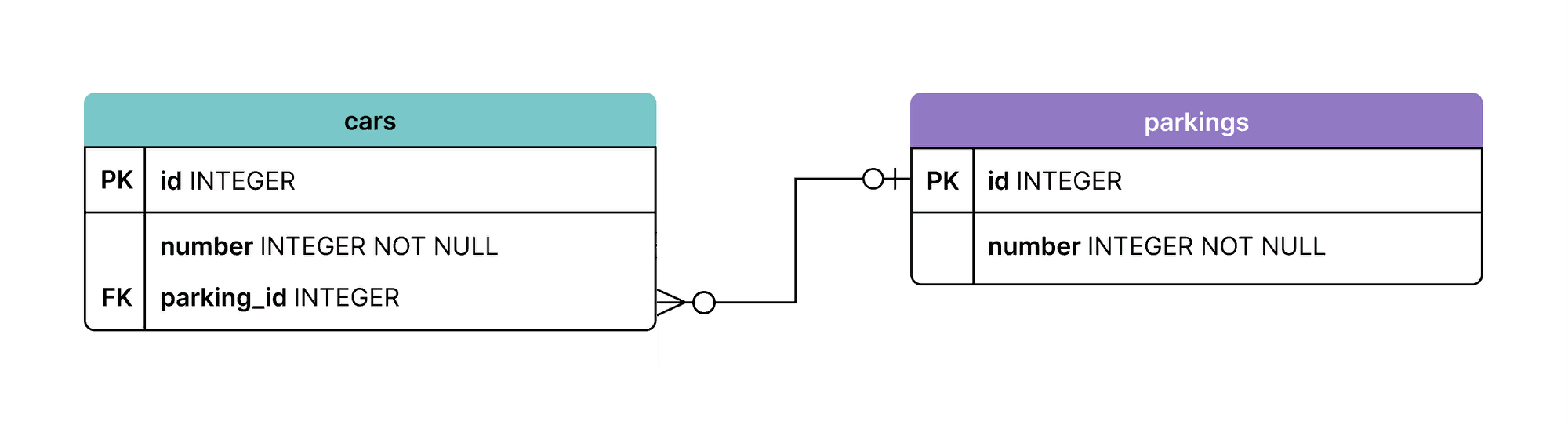
В школе обязательно есть классы — помещения для занятий. Классов в школе может быть много; каждый определённый класс обязательно расположен в какой-то школе, он не может существовать сам по себе. Но и в каждой школе обязательно должна быть хотя бы одна классная комната.

Связь обязательна: каждая запись таблицы **classrooms** должна ссылаться на запись в **schools**.



### Один-ко-многим: необязательная связь

Автомобильная стоянка может вмещать множество машин. Но может случиться и так, что на парковке нет ни одной машины. Связь между парковкой и автомобилями — один-ко-многим, необязательная.



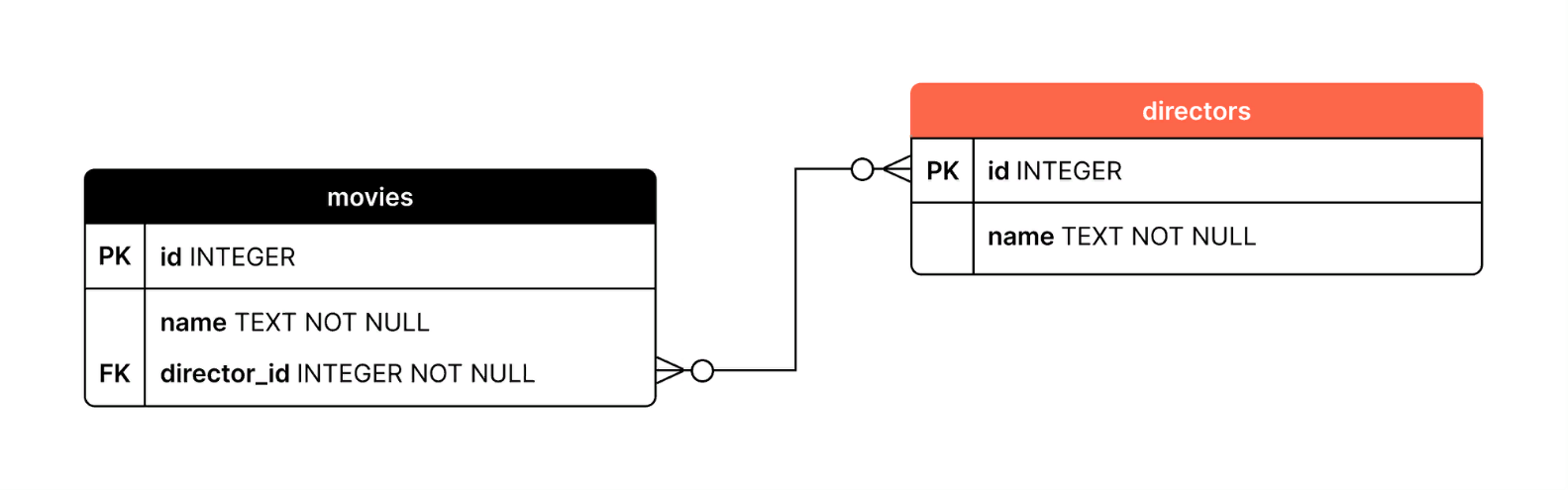
## Связь «многие-ко-многим» (M:M)

Это самый сложный тип связей: нескольким записям из одной таблицы могут соответствовать несколько записей из другой.

* **Учителя-предметники.** Один учитель может преподавать в нескольких классах, в то же самое время в одном классе может преподавать несколько учителей. Несколько учителей — несколько классов.
* **Фильмы — режиссёры.** Каждый режиссёр может снимать разные фильмы. У одного фильма может быть несколько режиссёров.

### Многие-ко-многим: обязательная связь

Режиссёр должен снять хотя бы один фильм (иначе какой же он режиссёр), и у фильма должен быть хотя бы один режиссёр. Режиссёр может снять много фильмов, а у одного фильма может быть несколько режиссёров.



Картинка выразительная, но это лишь упрощённое изображение связи: в одну ячейку FK нельзя записать несколько ссылок.

| **id** | **name** | **original\_name\_id** | **type\_id** | **director\_id** |
| --- | --- | --- | --- | --- |
| 6 | Она написала убийство | 2 | 4 | 5, 6, 7 |

В реальной таблице БД нельзя сослаться из одной ячейки сразу на несколько записей.

На уровне базы данных нельзя установить прямую связь M:M между двумя таблицами. Для такой связи необходима промежуточная таблица.

Название промежуточной таблицы принято составлять из имён тех таблиц, которые связаны через неё; в нашем случае получилось название **directors\_movies.**

Поля таблицы **directors\_movies**:

* movie\_id: ссылка на колонку id в movie.
* director\_id: ссылка на колонку id в таблице directors.

Посредством этой таблицы можно связать один фильм с несколькими режиссёрами и одного режиссёра — с несколькими фильмами.

При связи «многие-ко-многим» в связанных таблицах не нужно поле с внешним ключом; обратите внимание — в таблице **movies** нет никаких ссылок на таблицу **directors**.

**movies**

| **id** | **name** |
| --- | --- |
| 1 | Безумные Мелодии Луни Тюнз |
| 2 | Весёлые мелодии |
| 6 | Она написала убийство |

**directors**

| **id** | **name** |
| --- | --- |
| 1 | Текс Эйвери |
| 5 | Кори Аллен |
| 6 | Питер Фишер |
| 7 | Ричард Левинсон |

**directors\_movies**

| **movie\_id** | **director\_id** |
| --- | --- |
| 1 | 1 |
| 2 | 1 |
| 6 | 5 |
| 6 | 6 |
| 6 | 7 |

Связи в этой таблице можно прочитать так:

| **movie\_id** |
| --- |
| Безумные Мелодии Луни Тюнз — Текс Эйвери |
| Весёлые мелодии — Текс Эйвери |
| Она написала убийство — Кори Аллен |
| Она написала убийство — Питер Фишер |
| Она написала убийство — Ричард Левинсон |

При проектировании базы данных важно предусмотреть связи, которые могут потребоваться в дальнейшем. Например, если забыть о том, что у фильма может быть больше одного режиссёра и создать между фильмом и режиссёром связь «один-ко-многим» — в ближайшей перспективе придётся изменять базу данных. А если к тому моменту в таблице будет много записей — возникнут сложности.

Выберите соответствующий тип отношений:

|  |  |  |
| --- | --- | --- |
| Связь M:M | Связь 1:M | Связь 1:1 |
| Путешественник – страна (Человек может посетить любое количество стран. В страну могут приехать сколько угодно путешественников. N:M) | Страна – город  (Каждый город находится в одной стране, но в стране может быть много городов. 1:M) | Страна - офиц. Столица  (В каждой стране есть одна столица. Каждая столица принадлежит одной стране. 1:1) |
| Читатель-книги  (Один человек может читать разные книги, а каждую книгу могут прочесть несколько читателей. N:M) | Мама – ребенок  (У одной мамы может быть несколько детей, но у ребёнка —только одна мама. 1:M) | Автобус – водитель  (В одном автобусе может, конечно, ехать несколько водителей, но рулить будет один. Это связь 1:1) |
| Друг – друг  (У человека может быть несколько друзей; у каждого из друзей тоже может быть несколько друзей. N:M) | Человек – паспорт  (У каждого человека может быть несколько паспортов (загранпаспорт, второй гражданский паспорт), но у каждого паспорта может быть только один владелец. 1:M) |  |

## Композитный первичный ключ

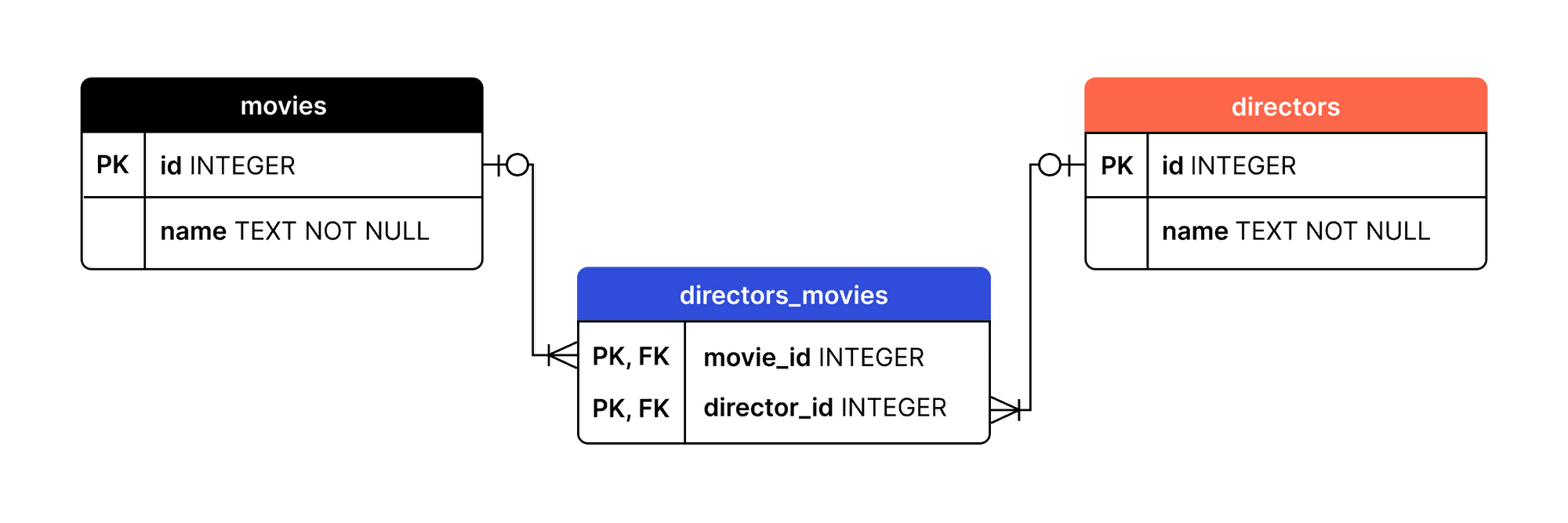
Присмотримся повнимательнее к промежуточной таблице **directors\_movies**.

Отдельные значения в колонках movie\_id и director\_id могут повторяться — так, в нашей таблице в колонке movie\_id повторяется значение **6** (ссылка на фильм «Она написала убийство»), а в колонке director\_id дважды встречается единица — ссылка на режиссёра Текса Эйвери.

| **Фильм** | **Режиссёр** |
| --- | --- |
| Безумные Мелодии Луни Тюнз | Текс Эйвери |
| Весёлые мелодии | Текс Эйвери |
| Она написала убийство | Кори Аллен |
| Она написала убийство | Питер Фишер |
| Она написала убийство | Ричард Левинсон |

А вот каждая пара значений movie\_id и director\_id должна быть уникальна: нет смысла дважды давать ссылку с одного и того же фильма на одного и того же режиссёра. **Пара значений этих полей служит первичным ключом**.

На ER-диаграмме обозначено, что первичным ключом в таблице **directors\_movies** служат сразу два поля:



Первичные ключи, состоящие из значений нескольких полей, называют **композитными первичными ключами**.

Если в таблице несколько полей обозначены как Primary key — значит, в таблице применяется композитный первичный ключ. Именно поэтому поля таблицы **directors\_movies** одновременно обозначены как **FK** (это ссылки на другие таблицы) и как **PK** — они составляют композитный первичный ключ.

Другим примером композитного первичного ключа может быть пара «серия-номер» паспорта. Ни серия, ни номер не обязаны быть уникальными, но в паре эти значения составляют уникальный идентификатор, определяющий документ и его владельца.

Чем отличаются первичный и внешний ключи?

- Внешний ключ отвечает за связи между таблицами и может повторяться. Первичный ключ тоже играет роль в связях, но ему важно быть уникальным.

### Как рисовать **ER-диаграммы**

Для визуализации отношений в базах данных есть много инструментов, в том числе — бесплатных и доступных онлайн. Например, на сайте [quickdatabasediagrams.com](https://www.quickdatabasediagrams.com/).

На практике junior-разработчики обычно читают диаграммы, а не рисуют их.

**12/17 Отношения между таблицами. Один-к-одному**

На схемах всё выглядит неплохо, пора поэкспериментировать со связями в коде.

Эксперимент первый: свяжем каждый фильм с его оригинальным названием, которое хранится в другой таблице.

Создадим новую БД с двумя таблицами — **movies** и **original\_names**.

В таблице **movies** создадим три поля:

id — primary key;

name — название фильма (обязательное поле);

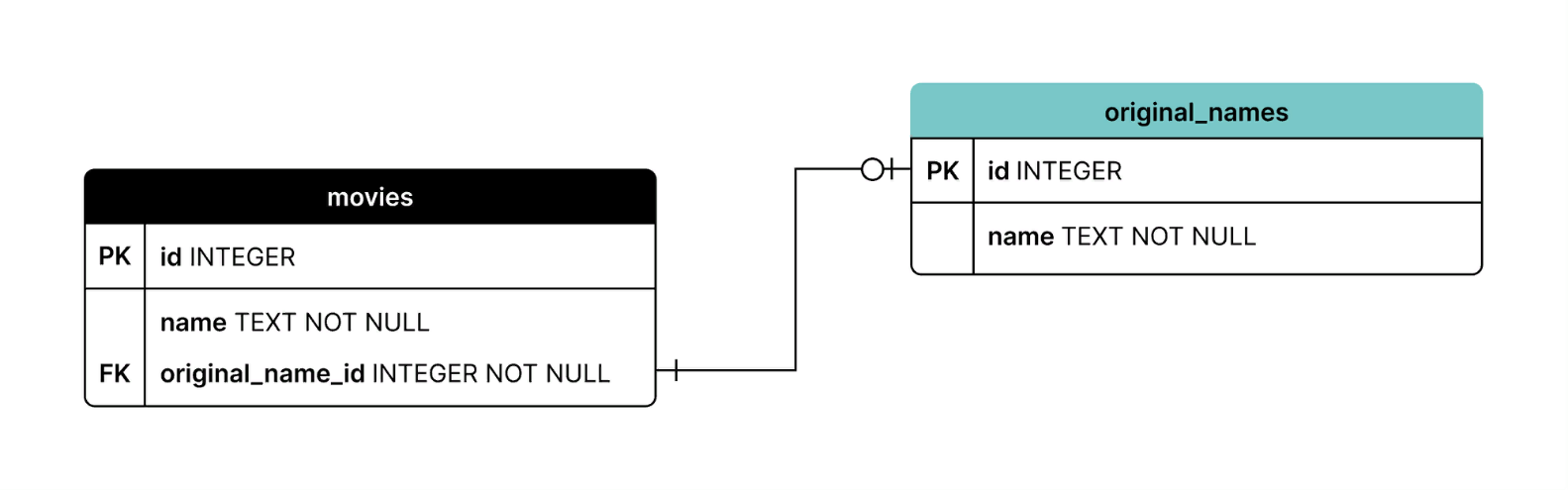
original\_name\_id — foreign key; через это поле таблица **movies** будет связана с таблицей **original\_names.**

В таблице **original\_names** будет два поля:

id — primary key;

name — оригинальное название фильма (обязательное поле);

Нужно связать записи в таблицах; у каждого фильма обязательно есть оригинальное название.



*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.executescript('''*

*CREATE TABLE IF NOT EXISTS original\_names(*

*id INTEGER PRIMARY KEY,*

*name TEXT NOT NULL*

*);*

*CREATE TABLE IF NOT EXISTS movies(*

*id INTEGER PRIMARY KEY,*

*name TEXT NOT NULL,*

*original\_name\_id INTEGER NOT NULL UNIQUE,*

*FOREIGN KEY(original\_name\_id) REFERENCES original\_names(id)*

*);*

*''')*

*con.commit()*

*con.close()*

Вот как создаётся связь:

1. В таблице movies создаётся поле original\_name\_id типа INTEGER (только числа).
2. UNIQUE объявляет это поле уникальным в пределах таблицы: в ячейках этой колонки не может быть двух одинаковых значений — ведь два разных фильма не могут ссылаться на одно и то же оригинальное название.
3. NOT NULL — поле не может быть пустым, при создании каждой записи его обязательно нужно заполнять.
4. FOREIGN KEY(original\_name\_id) — объявляет поле original\_name\_id внешним ключом.
5. Ключевое слово REFERENCES указывает с каким полем связан ключ. В нашем случае это поле id в таблице original\_names.

Заполним таблицы. Для этого сопоставим русские названия фильмов с оригинальными:

| **Русские названия** | **Оригинальные названия** |
| --- | --- |
| Безумные Мелодии Луни Тюнз | Looney Tunes |
| Весёлые мелодии | Merrie Melodies |
| Кто подставил кролика Роджера | Who Framed Roger Rabbit |
| Хороший, плохой, злой | Il Buono, il brutto, il cattivo |
| Последний киногерой | Last Action Hero |
| Она написала убийство | Murder, She Wrote |

В таблице **original\_names** фильмы могут идти в другом порядке, чем в таблице **movies**, и храниться под другими id. При заполнении таблицы **movies** важно указать правильный original\_name\_id, чтобы ссылка вела на нужную запись в таблице **original\_names**.

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*original\_names = [*

*(1, 'Last Action Hero'),*

*(2, 'Murder, She Wrote'),*

*(3, 'Looney Tunes'),*

*(4, 'Il Buono, il brutto, il cattivo'),*

*(5, 'Who Framed Roger Rabbit'),*

*(6, 'Merrie Melodies')*

*]*

*movies = [*

*(1, 'Безумные Мелодии Луни Тюнз', 3),*

*(2, 'Весёлые мелодии', 6),*

*(3, 'Кто подставил кролика Роджера', 5),*

*(4, 'Хороший, плохой, злой', 4),*

*(5, 'Последний киногерой', 1),*

*(6, 'Она написала убийство', 2)*

*]*

*cur.executemany('INSERT INTO original\_names VALUES(?, ?);', original\_names)*

*cur.executemany('INSERT INTO movies VALUES(?, ?, ?);', movies)*

*con.commit()*

*con.close()*

В результате выполнения этого кода будут созданы две таблицы, связанные по типу «один-к-одному».

### Получение информации из связанных таблиц

Выборку из таблиц, связанных 1:1, можно получить, например, таким запросом:

*...*

*cur.execute('''*

*-- Вернуть поле name из таблицы movies и поле name из original\_names*

*SELECT movies.name,*

*original\_names.name*

*-- ...из двух таблиц*

*FROM movies,*

*original\_names*

*-- Выводить только те значения полей, для которых верно условие*

*WHERE movies.original\_name\_id = original\_names.id*

*''')*

*for result in cur:*

*print(result)*

*...*

Результат:

*('Безумные Мелодии Луни Тюнз', 'Looney Tunes')*

*('Весёлые мелодии', 'Merrie Melodies')*

*('Кто подставил кролика Роджера', 'Who Framed Roger Rabbit')*

*('Хороший, плохой, злой', 'Il Buono, il brutto, il cattivo')*

*('Последний киногерой', 'Last Action Hero')*

*('Она написала убийство', 'Murder, She Wrote')*

Вот дословный перевод запроса: «из таблиц **movies** и **original\_names** верни столбцы movies.name и original\_names.name и покажи только те пары полей, для которых выполняется условие movies.original\_name\_id = original\_names.id».

При выборке из нескольких таблиц нужно указывать не только имя поля, но и имя таблицы, чтобы СУБД точно знала, о каком поле идёт речь.

Длинным названиям столбцов лучше давать короткие псевдонимы с помощью оператора AS:

*SELECT movies.name AS translation,*

*original\_names.name AS original*

*FROM movies,*

*original\_names*

*WHERE*

*movies.original\_name\_id = original\_names.id*

*-- Для интереса добавим условие*

*AND*

*original LIKE 'M%'*

1. Задача

В запрос, который создаёт таблицу **ice\_cream**, добавьте создание связанной таблицы **wrappers** — в ней будут храниться названия обёрток для мороженого.

В таблице **wrappers** должны быть два поля. Название обёртки должно храниться в текстовом поле name.

Каждая запись из таблицы **ice\_cream** может ссылаться на одну из записей в таблице **wrappers**: у каждого мороженого может быть одна уникальная обёртка. Два разных мороженых не могут ссылаться на одну обёртку.

У некоторых сортов мороженого может и не быть обёртки, так что связь — не обязательна; в таблице **ice\_cream** значение поля, ссылающегося на таблицу **wrappers**, может быть NULL.

* Напишите запрос, создающий таблицу **wrappers**.
* В таблице **wrappers** обязательно создайте числовое поле id и обозначьте его как первичный ключ таблицы, иначе не получится сослаться на записи этой таблицы.
* В запрос на создание таблицы **ice\_cream** добавьте поле, в котором будет указана связь с таблицей **wrappers.**
* Если поле ссылается на другую таблицу — связь нужно описать с помощью ключевого слова REFERENCES.
* Чтобы указать, что поле не может быть пустым — используйте инструкцию NOT NULL.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.executescript('''*

*CREATE TABLE IF NOT EXISTS*

*wrappers(*

*id INTEGER PRIMARY KEY,*

*name TEXT NOT NULL);*

*CREATE TABLE IF NOT EXISTS*

*ice\_cream(*

*id INTEGER PRIMARY KEY,*

*name TEXT,*

*description TEXT,*

*wrapper\_id INTEGER NOT NULL*

*UNIQUE,*

*FOREIGN KEY(wrapper\_id)*

*REFERENCES wrappers(id)*

*);*

*''')*

*con.commit()*

*con.close()*

2. Задача

В предыдущей задаче вы создали БД, а мы уже наполнили её данными.

Сделайте запрос к связанным таблицам и получите выборку вида название мороженого, название обёртки.

В выборке должны быть только те сорта мороженого, обёртка которых называется «Бумажная с черепами».

* Напишите запрос к таблице **ice\_cream** и связанной с ней таблице **wrappers**
* Название обёртки хранится в поле name таблицы **wrappers**. Отфильтруйте записи с помощью WHERE.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.execute('''*

*SELECT ice\_cream.name,*

*wrappers.name*

*FROM ice\_cream,*

*wrappers*

*WHERE ice\_cream.wrapper\_id = wrappers.id*

*AND wrappers.name = 'Бумажная с черепами';*

*''')*

*for result in cur:*

*print(result)*

*con.close()*

Результат:

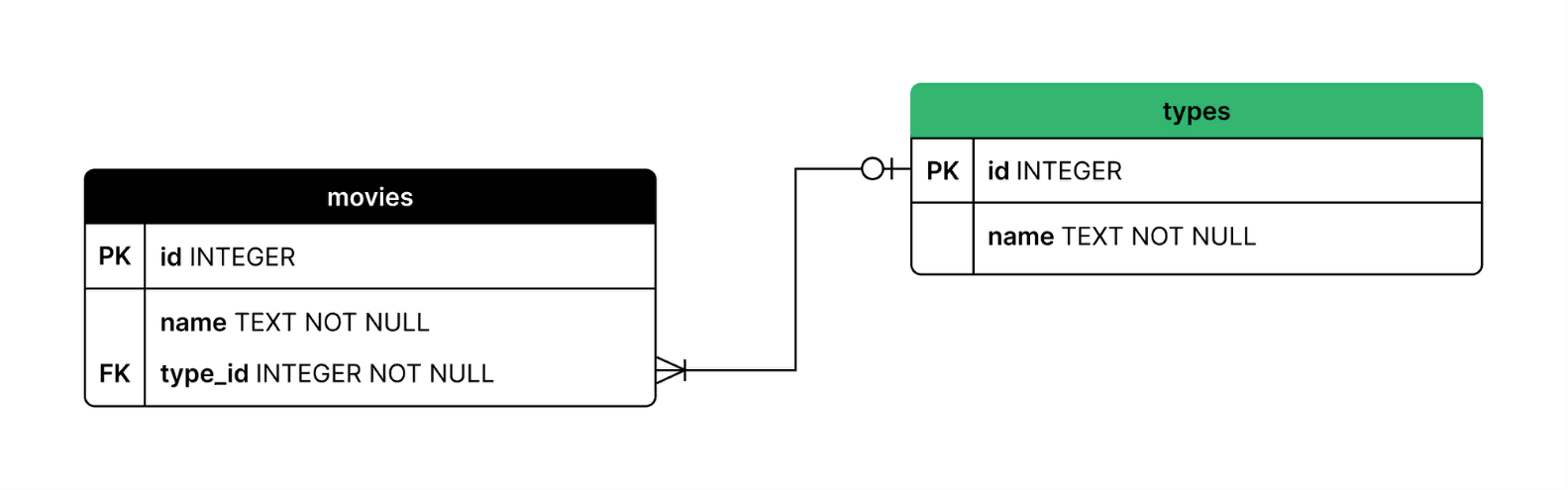
('Томатное мороженое', 'Бумажная с черепами')

**13/17 Отношения между таблицами. Один-ко-многим**

Продолжим эксперименты с таблицами и связями между ними. В нашей фильмотеке каждое произведение может относиться к одному из типов: мультфильм, фильм, сериал или мультсериал.

| **Название** | **Тип** |
| --- | --- |
| Безумные Мелодии Луни Тюнз | Мультсериал |
| Весёлые мелодии | Мультсериал |
| Кто подставил кролика Роджера | Фильм |
| Хороший, плохой, злой | Фильм |
| Последний киногерой | Фильм |
| Она написала убийство | Сериал |

Перечень типов фильмов сохраним в таблице **types**; к этой таблице должна «тянуться» связь от **movies**. Не будем перегружать таблицу **movies** лишними полями: забудем на время про поле original\_name\_id и сосредоточимся на связи с **types**. Для этого в таблице **movies** создадим FK-поле type\_id.



Кинолента может относиться только к одному типу, и для любого фильма обязательно должен быть указан его тип — значение поля type\_id не может быть NULL. С другой стороны — к одному типу может относиться много фильмов из таблицы **movies**. Это связь «один-ко-многим».

В SQL создание такой связи описывается так:

*CREATE TABLE IF NOT EXISTS movies(*

*...*

*type\_id INTEGER NOT NULL,*

*FOREIGN KEY(type\_id) REFERENCES types(id)*

*...);*

В переводе на русский:

* создай поле обязательное type\_id целочисленного типа; это поле не должно быть пустым;
* объяви это поле внешним ключом, связанным с первичным ключом другой таблицы — с полем id таблицы **types**.\*.

Для поля type\_id не указан параметр UNIQUE; таким образом, несколько записей из **movies** могут ссылаться на одну и ту же запись в **types**.

В Python создание связанных таблиц **movies** и **types** будет выглядеть так:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.executescript('''*

*CREATE TABLE IF NOT EXISTS types(*

*id INTEGER PRIMARY KEY,*

*name TEXT NOT NULL*

*);*

*CREATE TABLE IF NOT EXISTS movies(*

*id INTEGER PRIMARY KEY,*

*name TEXT NOT NULL,*

*type\_id INTEGER NOT NULL,*

*FOREIGN KEY(type\_id) REFERENCES types(id)*

*);*

*''')*

*con.commit()*

*con.close()*

Заполним таблицы данными; следим, чтобы не перепутать значения полей type\_id в **movies**:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*movies = [*

*(1, 'Безумные Мелодии Луни Тюнз', 2),*

*(2, 'Весёлые мелодии', 2),*

*(3, 'Кто подставил кролика Роджера', 3),*

*(4, 'Хороший, плохой, злой', 3),*

*(5, 'Последний киногерой', 3),*

*(6, 'Она написала убийство', 4),*

*]*

*types = [*

*(1, 'Мультфильм'),*

*(2, 'Мультсериал'),*

*(3, 'Фильм'),*

*(4, 'Сериал'),*

*]*

*cur.executemany('INSERT INTO types VALUES(?, ?);', types)*

*cur.executemany('INSERT INTO movies VALUES(?, ?, ?);', movies)*

*con.commit()*

*con.close()*

Выбрать все названия кинолент из категории 'Фильм' можно таким запросом:

*SELECT movies.name,*

*types.name*

*FROM movies,*

*types*

*WHERE movies.type\_id = types.id AND types.name = 'Фильм';*

1.Задача

Сорта мороженого могут разделяться на группы — категории. Например, какие-то сорта войдут в категорию «Экзотические», какие-то — в категорию «Классические»; возможно, потом будет и категория «Не вздумай пробовать».

В каждой категории может быть несколько сортов мороженого, но каждое мороженое может принадлежать лишь одной категории: это классический пример связи 1:M.

Нужно добавить категории в базу данных.

Допишите код SQL-запроса:

1. Добавьте запрос, который создаст таблицу **categories** — в ней будут храниться типы мороженого. В таблице должно быть два обязательных поля:
   * id — уникальный идентификатор записи,
   * slug — обязательное текстовое поле; значение этого поля будет использоваться, например, для создания URL страницы, где будут отображаться все сорта из одной категории.
2. Свяжите таблицу **ice\_cream** с таблицей **categories**: добавьте в **ice\_cream** поле category\_id, ссылающееся на PK в таблице **categories**. Связь — обязательна: не может быть мороженого без категории.

* Допишите в запрос создание таблицы **categories**.
* В запросе, создающем таблицу **ice\_cream**, опишите поле category\_id.
* Поле, ссылающееся на другую таблицу, описывается через ключевые слова FOREIGN KEY… REFERENCES…

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.executescript('''*

*CREATE TABLE IF NOT EXISTS*

*categories(*

*id INTEGER PRIMARY KEY,*

*slug TEXT NOT NULL);*

*CREATE TABLE IF NOT EXISTS*

*ice\_cream(*

*id INTEGER PRIMARY KEY,*

*name TEXT NOT NULL,*

*price REAL NOT NULL,*

*category\_id INTEGER NOT NULL*

*UNIQUE,*

*FOREIGN KEY(category\_id)*

*REFERENCES categories(id) );*

*''')*

*con.commit()*

*con.close()*

2.Задача

Базу данных, которую вы создали, мы наполнили информацией. Получите из базы самые дорогие сорта мороженого в каждой из категорий. Выборка должна выглядеть так:

*(название\_мороженого, слаг\_категории, цена)*

Выборка должна быть отсортирована по цене, от дорогого к дешёвому.

* Напишите запрос к таблицам **ice\_cream** и **categories.**
* Примените агрегигрующую функцию MAX к полю ice\_cream.price.
* Решить задачу помогут GROUP BY и ORDER BY.
* Вот так должен выглядеть ответ:

*('Мороженое с кузнечиками', 'exotic', 688.0)*

*('Пломбир 1937', 'normal', 78.0)*

Код:

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.execute('''*

*SELECT ice\_cream.name,*

*categories.slug,*

*MAX(ice\_cream.price)*

*FROM ice\_cream, categories*

*WHERE ice\_cream.category\_id = categories.id*

*GROUP BY categories.slug*

*ORDER BY ice\_cream.price DESC*

*''')*

*for result in cur:*

*print(result)*

*con.commit()*

*con.close()*

Результат:

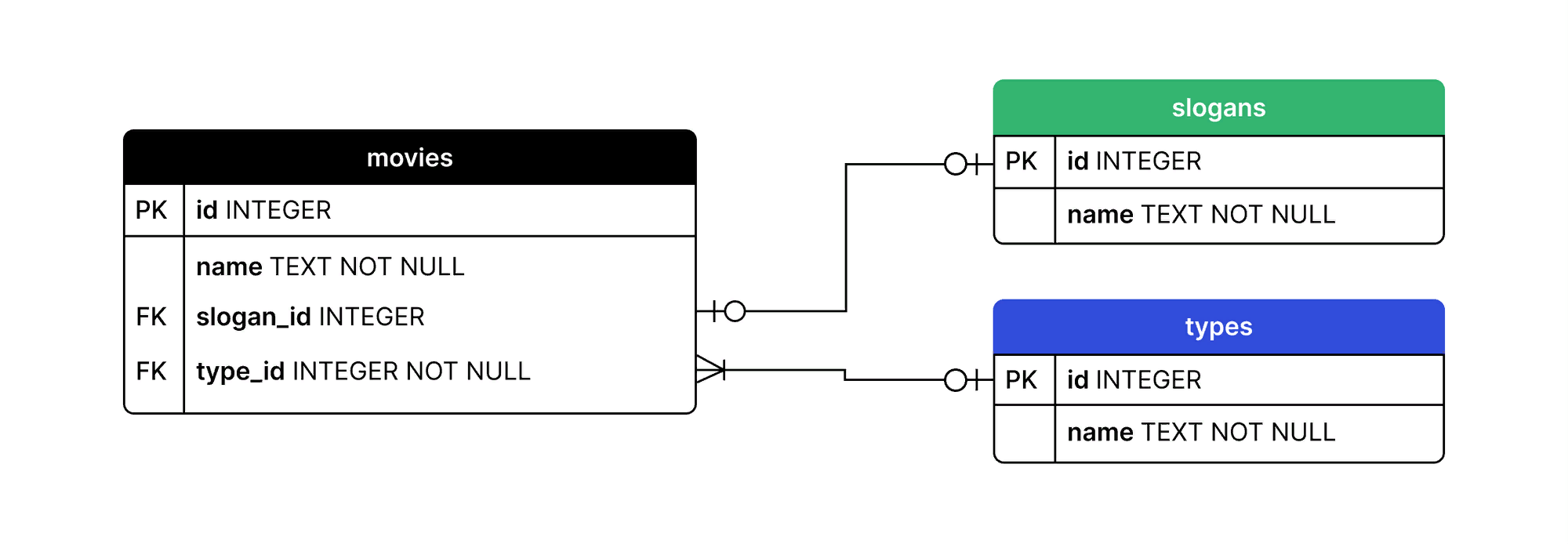
('Мороженое с кузнечиками', 'exotic', 688.0)

('Пломбир 1937', 'normal', 78.0)

**14/17 Объединение таблиц: JOIN**

Продолжим наращивать базу данных с фильмами: соберём в неё таблицы **movies** и **types**; добавим таблицу **slogans** (рекламный девиз фильма или ударная фраза из него). Записи из таблицы **movies** будут ссылаться на слоганы через необязательную связь один-к-одному: у фильма может и не быть слогана.

Вот ER-диаграмма базы данных:



**movies**

| **id** | **name** | **type\_id** | **slogan\_id** |
| --- | --- | --- | --- |
| 1 | Безумные Мелодии Луни Тюнз | 1 | NULL |
| 2 | Весёлые мелодии | 1 | NULL |
| 3 | Хороший, плохой, злой | 2 | 1 |
| 4 | Последний киногерой | 2 | 2 |
| 5 | Она написала убийство | 3 | 3 |

**slogans**

| **id** | **name** |
| --- | --- |
| 1 | For Three Men The Civil War Wasn't Hell. It Was Practice! |
| 2 | This isn't the movies anymore |
| 3 | Tonight on Murder She Wrote |
| 4 | I’ll be back |

**types**

| **id** | **name** |
| --- | --- |
| 1 | Мультсериал |
| 2 | Фильм |
| 3 | Сериал |
| 4 | Мультфильм |

[Вот эта база](https://code.s3.yandex.net/backend-developer/learning-materials/delugov/db.sqlite), берите, подключайтесь и экспериментируйте.

Для начала посмотрим, что вернёт запрос одновременно к двум таблицам:

*SELECT \**

*FROM movies, slogans;*

Запрос создал результирующую таблицу, со столбцами

*movies.id | movies.name | movies.type\_id | movies.slogan\_id | slogans.id | slogans.name*

(1, 'Безумные Мелодии Луни Тюнз', 1, None, 1, "For Three Men The Civil War Wasn't Hell. It Was Practice!")

(1, 'Безумные Мелодии Луни Тюнз', 1, None, 2, "This isn't the movies anymore")

(1, 'Безумные Мелодии Луни Тюнз', 1, None, 3, 'Tonight on Murder She Wrote')

(1, 'Безумные Мелодии Луни Тюнз', 1, None, 4, "I'll be back")

(2, 'Весёлые мелодии', 1, None, 1, "For Three Men The Civil War Wasn't Hell. It Was Practice!")

(2, 'Весёлые мелодии', 1, None, 2, "This isn't the movies anymore")

(2, 'Весёлые мелодии', 1, None, 3, 'Tonight on Murder She Wrote')

(2, 'Весёлые мелодии', 1, None, 4, "I'll be back")

(3, 'Хороший, плохой, злой', 2, 1, 1, "For Three Men The Civil War Wasn't Hell. It Was Practice!")

(3, 'Хороший, плохой, злой', 2, 1, 2, "This isn't the movies anymore")

(3, 'Хороший, плохой, злой', 2, 1, 3, 'Tonight on Murder She Wrote')

(3, 'Хороший, плохой, злой', 2, 1, 4, "I'll be back")

(4, 'Последний киногерой', 2, 2, 1, "For Three Men The Civil War Wasn't Hell. It Was Practice!")

(4, 'Последний киногерой', 2, 2, 2, "This isn't the movies anymore")

(4, 'Последний киногерой', 2, 2, 3, 'Tonight on Murder She Wrote')

(4, 'Последний киногерой', 2, 2, 4, "I'll be back")

(5, 'Она написала убийство', 1, 3, 1, "For Three Men The Civil War Wasn't Hell. It Was Practice!")

(5, 'Она написала убийство', 1, 3, 2, "This isn't the movies anymore")

(5, 'Она написала убийство', 1, 3, 3, 'Tonight on Murder She Wrote')

(5, 'Она написала убийство', 1, 3, 4, "I'll be back")

В результирующую выборку включены все поля объединяемых таблиц; **к каждой** записи из **movies** присоединена **каждая** запись из **slogans**. На математическом языке это называется «декартово произведение множеств» (или [«прямое произведение множеств»](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D1%8F%D0%BC%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5)).

Выглядит не очень практично, но ведь эту выборку можно отфильтровать!

Поставим задачу: получить из таблицы **movies** все записи; для каждой из записей вывести слоган, привязанный к этой записи.

Отфильтруем декартово произведение с помощью WHERE. Оставим только те строки, в которых значение поля slogan\_id в таблице **movies** равно значению id в таблице **slogans** :

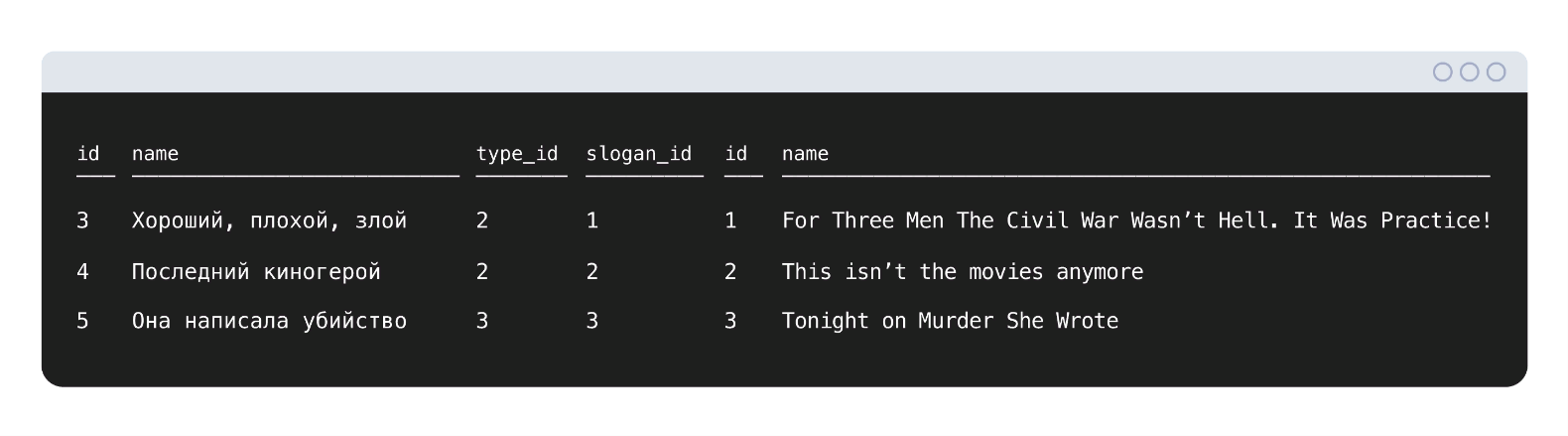
*SELECT \**

*FROM movies,*

*slogans*

*WHERE movies.slogan\_id = slogans.id;*

Результирующая выборка будет такой:



Напечатаем результат из Python:

(3, 'Хороший, плохой, злой', 2, 1, 1, "For Three Men The Civil War Wasn't Hell. It Was Practice!")

(4, 'Последний киногерой', 2, 2, 2, "This isn't the movies anymore")

(5, 'Она написала убийство', 3, 3, 3, 'Tonight on Murder She Wrote')

Удача. Успех. Хорошо!

Для пущей красоты можно показать только полезные столбцы:

*SELECT movies.name,*

*slogans.name*

*FROM movies,*

*slogans*

*WHERE movies.slogan\_id = slogans.id;*

## JOIN

Конструкция FROM movies, slogans соединяет таблицы, а WHERE — фильтрует получившуюся выборку.

По стандарту SQL92 принято отделять фильтрацию от условий соединения таблиц с помощью оператора JOIN (англ. «соединение»):

*-- Верни все поля*

*SELECT \**

*-- из таблицы movies*

*FROM movies*

*-- ...но перед этим присоедини таблицу slogans так, чтобы в записях*

*-- совпадали значения полей movies.slogan\_id и slogans.id*

*JOIN slogans ON movies.slogan\_id = slogans.id;*

Результат:

(3, 'Хороший, плохой, злой', 2, 1, 1, "For Three Men The Civil War Wasn't Hell. It Was Practice!")

(4, 'Последний киногерой', 2, 2, 2, "This isn't the movies anymore")

(5, 'Она написала убийство', 3, 3, 3, 'Tonight on Murder She Wrote')

Результат — тот же, а запрос выглядит лучше. Мухи отдельно — котлеты отдельно.

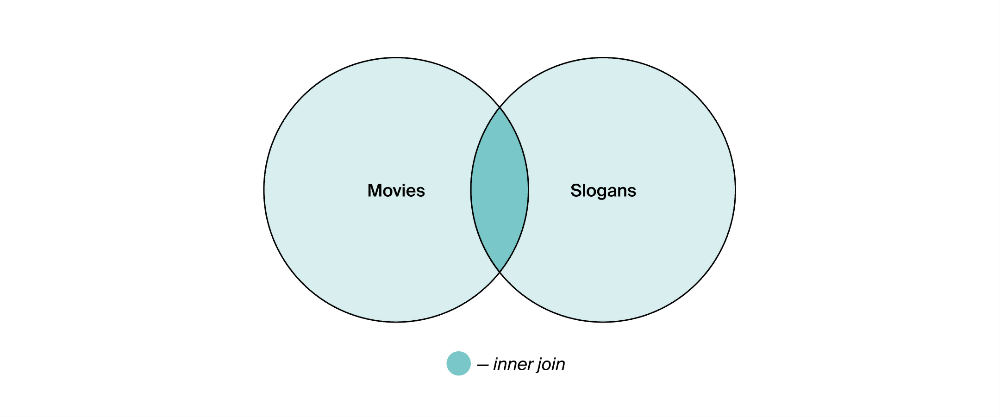
* Соединение и фильтрация разделены; это упрощает понимание запроса.
* Условия соединения для каждой пары таблиц содержатся в блоке ON, это уменьшает вероятность ошибки.

В классическом SQL существует пять типов JOIN; SQLite поддерживает только три из них. Но с помощью ловкости и смекалки неподдерживаемые типы можно заменить другими запросами.

### Внутреннее пересечение: INNER JOIN

Первый тип мы уже продемонстрировали. В примере выше был как раз **INNER JOIN** (или просто **JOIN**, без титулов).

Объединение таблиц через **INNER JOIN** можно представить схематически:



Запрос через JOIN можно сделать и к трём таблицам:

*SELECT movies.name,*

*slogans.name,*

*types.name*

*FROM movies*

*JOIN slogans ON movies.slogan\_id = slogans.id*

*JOIN types ON movies.type\_id = types.id;*

В этой выборке всего три фильма: мультики в неё не попали, ведь у них не указана связь с таблицей **slogans**.

Оператор INNER JOIN включает в результирующую таблицу только те записи, в которых выполняется условие, заданное в ON.

### Левое внешнее соединение: **LEFT OUTER JOIN**

При обработке запроса LEFT OUTER JOIN объединяемые таблицы условно называют **«левая»** и **«правая»**. «Левая» — та, которая вызвана в блоке FROM, «правая» — та, что указана после ключевого слова JOIN. «Правых» таблиц может быть и несколько.

В этом запросе **OUTER** — не обязательное слово. Можно использовать сокращённую запись: **LEFT JOIN.**

При выполнении запросов с **LEFT JOIN** возвращаются **все строки левой таблицы**. Данными из **правой таблицы** дополняются только те строки левой таблицы, для которых выполняются условия соединения, описанные после оператора ON. Для недостающих данных вместо строк правой таблицы вставляются NULL-значения.

*SELECT movies.name,*

*slogans.name*

*FROM movies*

*LEFT JOIN slogans ON movies.slogan\_id = slogans.id;*

Результат запроса:

('Безумные Мелодии Луни Тюнз', None)

('Весёлые мелодии', None)

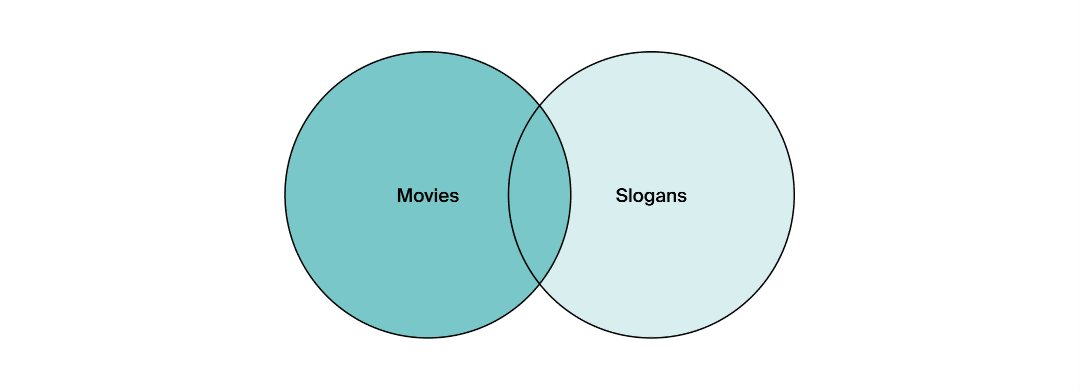
('Хороший, плохой, злой', "For Three Men The Civil War Wasn't Hell. It Was Practice!")

('Последний киногерой', "This isn't the movies anymore")

('Она написала убийство', 'Tonight on Murder She Wrote')

«Безумные Мелодии Луни Тюнз» и «Весёлые мелодии» остались в выборке, поскольку LEFT JOIN **возвращает все записи левой таблицы без исключения**.

Графически эта связь изображается так:



### **RIGHT OUTER JOIN**

**RIGHT JOIN** – это такое же объединение, как и **LEFT JOIN**, но **выводятся все записи из правой таблицы**, а к ним добавляются только те данные из левой таблицы, в которых есть ключ объединения.

В большинстве СУБД (но не в SQLite) можно отправить такой запрос:

SELECT movies.name,

types.name

FROM movies

RIGHT JOIN types ON movies.type\_id = types.id;

…и в ответ вернётся ответ

('Безумные Мелодии Луни Тюнз', 'Мультсериал')

('Весёлые мелодии', 'Мультсериал')

('Последний киногерой', 'Фильм')

('Хороший, плохой, злой', 'Фильм')

('Она написала убийство', 'Сериал')

(None, 'Мультфильм')

Будут выведены все записи из правой таблицы (из **types**) и связанные с ними записи из левой (из **movies**). В таблице **movies** нет ни одной записи, ссылающейся на тип «Мультфильм», поэтому вместо значения movies.name в этой строке стоит None.

В SQLite при попытке выполнить **RIGHT JOIN** возникнет ошибка: эта СУБД не поддерживает такие запросы.

*sqlite3.OperationalError: RIGHT and FULL OUTER JOINs are not currently supported*

Выход есть: можно применить «разрешённый» в SQLite LEFT JOIN, поменяв таблицы местами. В нашем случае нужно назвать **types** главной (левой), а **movies** присоединить через LEFT JOIN.

*SELECT movies.name,*

*types.name*

*FROM types*

*LEFT JOIN movies ON movies.type\_id = types.id;*

Результат этого запроса будет тем же, что и в предыдущем примере:

('Безумные Мелодии Луни Тюнз', 'Мультсериал')

('Весёлые мелодии', 'Мультсериал')

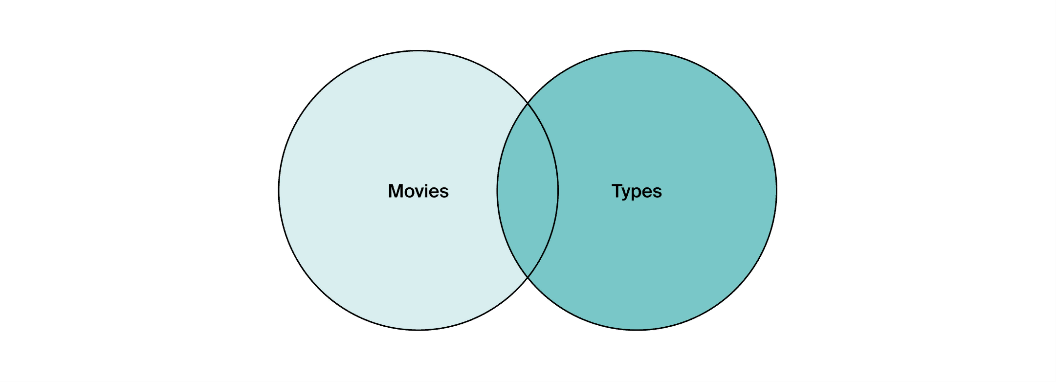
('Последний киногерой', 'Фильм')

('Хороший, плохой, злой', 'Фильм')

('Она написала убийство', 'Сериал')

(None, 'Мультфильм')

Графическое отображение **RIGHT JOIN**:



### **FULL OUTER JOIN**

При запросе **FULL (OUTER) JOIN** выводятся **все записи из объединяемых таблиц**. Те записи, у которых запрошенные значения совпадают — выводятся парами, у остальных недостающее значение заменяется на NULL (Python выведет None).

Другими словами, **FULL JOIN == LEFT JOIN + RIGHT JOIN.**

В классическом SQL сработает такой запрос:

*SELECT movies.name,*

*slogans.name*

*FROM movies*

*FULL JOIN slogans ON movies.slogan\_id = slogans.id;*

Результат запроса:

(None, "I'll be back")

('Безумные Мелодии Луни Тюнз', None)

('Весёлые мелодии', None)

('Она написала убийство', 'Tonight on Murder She Wrote')

('Последний киногерой', "This isn't the movies anymore")

('Хороший, плохой, злой', "For Three Men The Civil War Wasn't Hell. It Was Practice!")

SQLite не поддерживает оператор **FULL JOIN**, при попытке выполнить такой запрос вернётся уже знакомая ошибка:

*sqlite3.OperationalError: RIGHT and FULL OUTER JOINs are not currently supported*

В SQLite такую задачу можно решить так: выполнить два запроса LEFT JOIN и объединить их через команду UNION — она объединяет данные из нескольких результирующих таблиц в одну:

*SELECT movies.name,*

*slogans.name*

*FROM movies*

*LEFT JOIN slogans ON movies.slogan\_id = slogans.id*

*UNION*

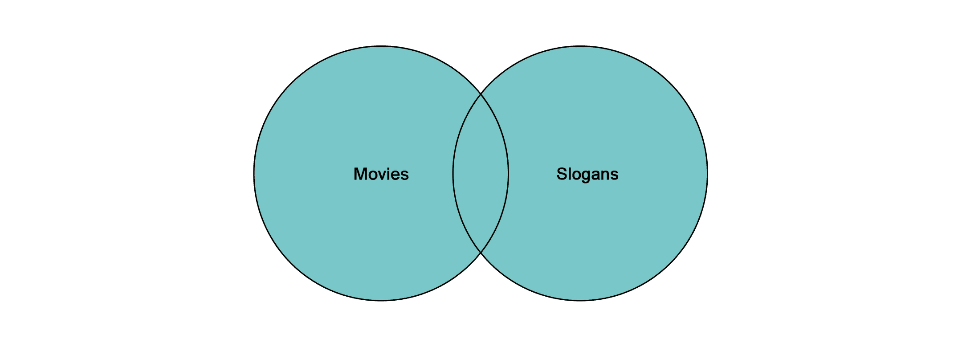
*SELECT movies.name,*

*slogans.name*

*FROM slogans*

*LEFT JOIN movies ON movies.slogan\_id = slogans.id;*

Графически **FULL JOIN** можно представить так:



### CROSS JOIN

Объединение таблиц через CROSS JOIN возвращает декартово произведение таблиц — каждая запись левой таблицы объединится с каждой записью правой. Параметр ON при запросах CROSS JOIN не применяется.

*SELECT movies.name,*

*slogans.name*

*FROM movies*

*CROSS JOIN slogans;*

Распечатаем результаты запроса:

('Безумные Мелодии Луни Тюнз', "For Three Men The Civil War Wasn't Hell. It Was Practice!")

('Безумные Мелодии Луни Тюнз', "This isn't the movies anymore")

('Безумные Мелодии Луни Тюнз', 'Tonight on Murder She Wrote')

('Безумные Мелодии Луни Тюнз', "I'll be back")

('Весёлые мелодии', "For Three Men The Civil War Wasn't Hell. It Was Practice!")

('Весёлые мелодии', "This isn't the movies anymore")

('Весёлые мелодии', 'Tonight on Murder She Wrote')

('Весёлые мелодии', "I'll be back")

('Хороший, плохой, злой', "For Three Men The Civil War Wasn't Hell. It Was Practice!")

('Хороший, плохой, злой', "This isn't the movies anymore")

('Хороший, плохой, злой', 'Tonight on Murder She Wrote')

('Хороший, плохой, злой', "I'll be back")

('Последний киногерой', "For Three Men The Civil War Wasn't Hell. It Was Practice!")

('Последний киногерой', "This isn't the movies anymore")

('Последний киногерой', 'Tonight on Murder She Wrote')

('Последний киногерой', "I'll be back")

('Она написала убийство', "For Three Men The Civil War Wasn't Hell. It Was Practice!")

('Она написала убийство', "This isn't the movies anymore")

('Она написала убийство', 'Tonight on Murder She Wrote')

('Она написала убийство', "I'll be back")

На практике CROSS JOIN применяется не очень часто, но и он может быть полезен. Например, если в одной таблице сохранён список жидкостей, а во второй — список возможной тары для расфасовки, от маленькой баночки до цистерны. В выборке получим все возможные комбинации «жидкость — тара».

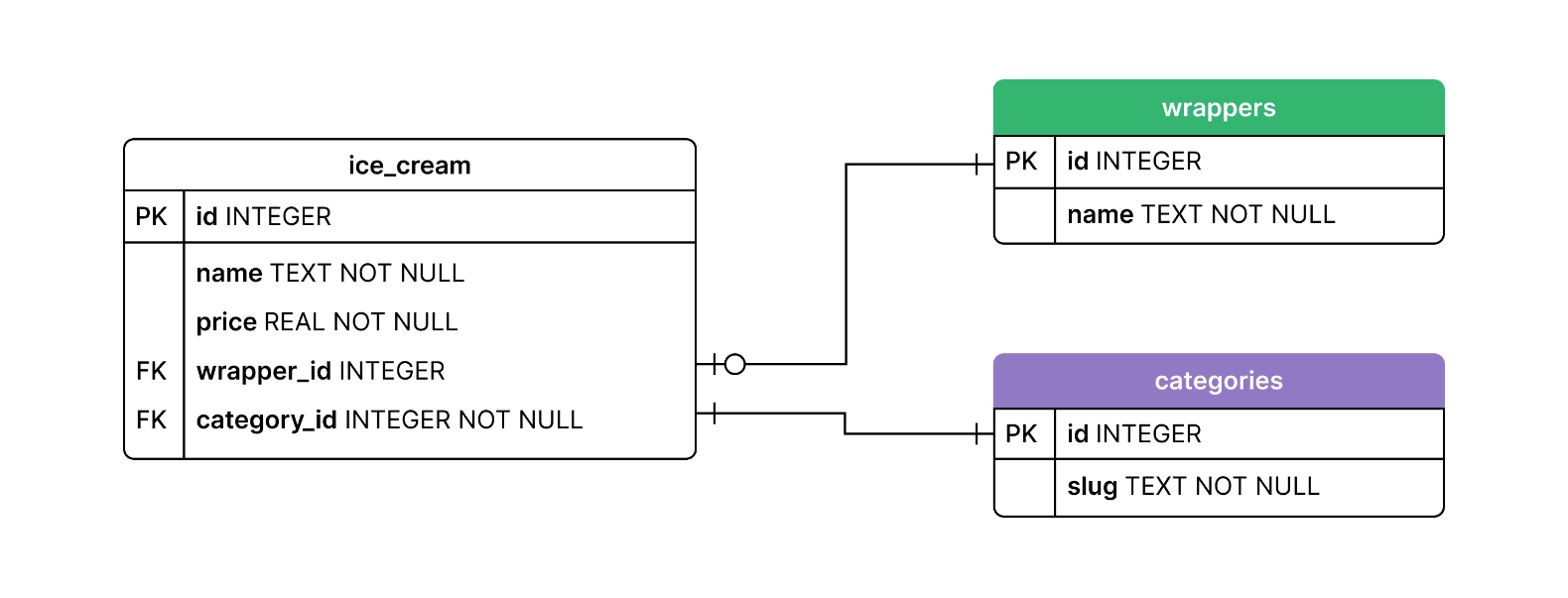
1. Задача

В тренажёре подготовлена база данных с тремя таблицами:

* **ice\_cream** — сорта мороженого;
* **categories** — категории мороженого;
* **wrappers** — перечень упаковок.

Таблицы **ice\_cream** и **wrappers** связаны 1:1 через поле ice\_cream.wrapper\_id; связь необязательна: у мороженого может и не быть упаковки.

Таблицы **ice\_cream** и **categories** связаны 1:1 через поле ice\_cream.category\_id; связь обязательна.



С помощью объединения таблиц выведите названия мороженого (поле **name** таблицы **ice\_cream**) и названия упаковок (поле **name** таблица **wrappers**) для тех сортов, у которых в описании упаковки используется слово «праздник» или однокоренные слова. В решении задачи поможет поиск по маске.

Сделайте запрос к таблице **ice\_cream** и присоедините таблицу **wrappers**: JOIN wrappers.

В условии ON укажите связь: ice\_cream.wrapper\_id = wrappers.id.

LIKE — поиск строки по шаблону; в шаблонах можно применять символы-маски: знак процента % заменяет любой **набор символов.** Символ % можно применять как **до**, так и **после** корня 'праздн'.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.execute('''*

*SELECT ice\_cream.name, wrappers.name*

*FROM ice\_cream*

*JOIN wrappers ON ice\_cream.wrapper\_id = wrappers.id*

*WHERE wrappers.name LIKE '%праздн%';*

*''')*

*for result in cur:*

*print(result)*

*con.commit()*

*con.close()*

Результат:

('Мороженое со вкусом сыра чеддер', 'Упаковка-оригами, незаменима для детских праздников')

('Пломбир 1937', 'Простая упаковка-салфетка с изображением Анфисы и праздничными поздравлениями')

2. Задача

Продолжим работать с той же базой:

Получите из базы данных информацию **о самом дешёвом мороженом** в каждой категории.

Выведите в консоль:

* название самого дешёвого мороженого в категории (поле name из таблицы **ice\_cream**);
* слаг категории, которой принадлежит это мороженое (поле slug из таблицы **categories**);
* название обёртки этого мороженого (поле name из таблицы **wrappers**);
* стоимость этого мороженого (поле price из таблицы **ice\_cream**);
* среднюю цену мороженого в его категории.

Описание мороженого в консоли должно выглядеть так:

*('название\_мороженого', 'slug\_категории', 'название\_упаковки', <стоимость>, <средняя\_стоимость\_в\_категории>)*

В БД две категории — значит, должно получиться две такие строки.

* Самое дешёвое мороженое получите, сгруппировав категории через GROUP BY и применив агрегирующую функцию MIN.
* За вычисление среднего арифметического отвечает агрегирующая функция AVG.
* Некоторые сорта мороженого не имеют обёртки, это может повлиять на получение правильного результата. Используйте LEFT JOIN с таблицей wrappers.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.execute('''*

*SELECT ice\_cream.name,*

*categories.slug,*

*wrappers.name,*

*MIN(ice\_cream.price),*

*AVG(ice\_cream.price)*

*FROM ice\_cream*

*LEFT JOIN wrappers ON ice\_cream.wrapper\_id=wrappers.id*

*JOIN categories ON ice\_cream.category\_id=categories.id*

*GROUP BY categories.id;*

*''')*

*for result in cur:*

*print(result)*

*con.commit()*

*con.close()*

Результат:

('Пивное мороженое', 'exotic', None, 500.0, 563.0)

('Томатное мороженое', 'normal', 'Бумажная с черепами', 0.1, 39.05)

# 15/17 Отношения между таблицами. Многие-ко-многим

У каждой картины есть автор. Но если в живописи два автора одной картины — это редкость, то несколько режиссёров одного фильма — это обычное дело. Публика хочет знать имена режиссёров, так что добавим их в нашу базу данных.

Для начала опишем условия:

* У фильма может быть один или несколько режиссёров.
* Режиссёр может снять один фильм или несколько фильмов.

Это классический пример связи M:M, «многие-ко-многим».

Соберём имена режиссёров в отдельную таблицу:

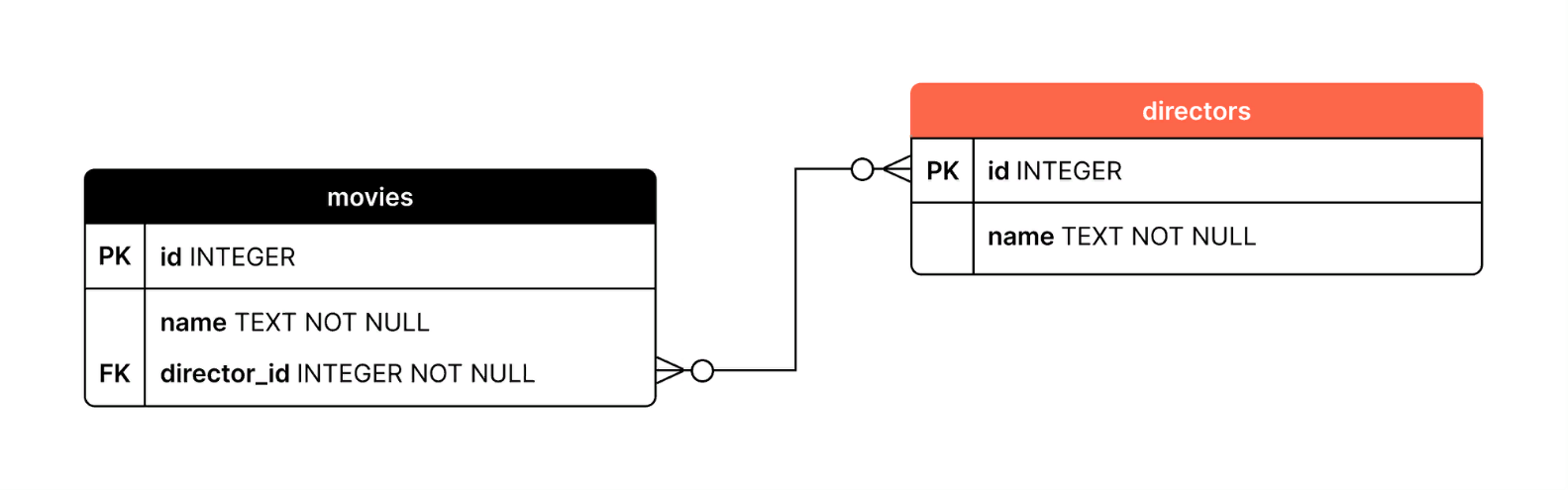
**directors**

| **id** | **name** |
| --- | --- |
| 1 | Текс Эйвери |
| 2 | Роберт Земекис |
| 3 | Серджо Леоне |
| 4 | Джон Мактирнан |
| 5 | Кори Аллен |
| 6 | Питер Фишер |
| 7 | Ричард Левинсон |

Сопоставим названия фильмов с id режиссёров:

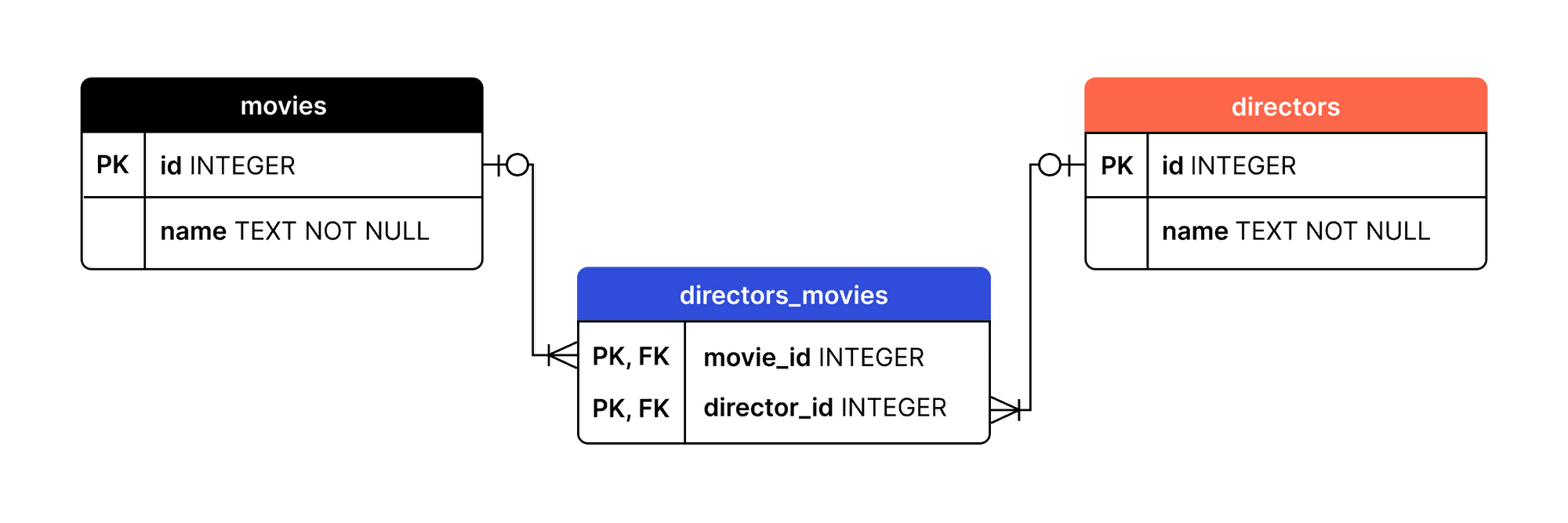
| **Название фильма** | **id режиссёра** |
| --- | --- |
| Безумные Мелодии Луни Тюнз | 1 |
| Весёлые мелодии | 1 |
| Кто подставил кролика Роджера | 2 |
| Хороший, плохой, злой | 3 |
| Последний киногерой | 4 |
| Она написала убийство | 5, 6, 7 |

Упрощённо эту связь можно отобразить на ER-диаграмме так: множественная связь к таблице **movies** и множественная — к таблице **directors**.



Но у фильма «Она написала убийство» целых три режиссёра, и чтобы указать их — в поле director\_id пришлось бы перечислить несколько id. Однако в одной ячейке типа INTEGER должно храниться только одно число.

На практике такую связь организуют через промежуточную таблицу: в ней сопоставляют PK записей из связанных таблиц.



Промежуточную таблицу назовём по именам таблиц, которые связываются через неё: **directors\_movies**.

Она содержит

* поле director\_id, которое хранит FK, ссылающийся на колонку id в таблице **directors**;
* поле movie\_id, которое хранит FK, ссылающийся на колонку id в таблице **movies**.

| **movie\_id** | **director\_id** |
| --- | --- |
| 1 | 1 |
| 2 | 1 |
| 6 | 5 |
| 6 | 6 |
| 6 | 7 |

Эта таблица связывает фильмы с режиссёрами:

| **Режиссёры** |
| --- |
| Безумные Мелодии Луни Тюнз — Текс Эйвери |
| Весёлые мелодии — Текс Эйвери |
| Она написала убийство — Кори Аллен |
| Она написала убийство — Питер Фишер |
| Она написала убийство — Ричард Левинсон |

Любой режиссер может снять один фильм только один раз. Ведь даже если он переснимет его спустя 10, 20, 1000 лет — это уже будет другой фильм. Поэтому здесь в качестве PRIMARY KEY используется сочетание (director\_id, movie\_id) — эта пара уникальна в пределах таблицы. Сочетание значений этих полей служит **композитным** первичным ключом.

Создать таблицу с композитным первичным ключом можно так:

*CREATE TABLE IF NOT EXISTS movies(*

*id INTEGER PRIMARY KEY,*

*name TEXT NOT NULL*

*);*

*CREATE TABLE IF NOT EXISTS directors(*

*id INTEGER PRIMARY KEY,*

*name TEXT NOT NULL*

*);*

*CREATE TABLE IF NOT EXISTS directors\_movies(*

*director\_id INTEGER NOT NULL,*

*movie\_id INTEGER NOT NULL,*

*-- Пару полей назначаем композитным первичным ключом:*

*PRIMARY KEY (director\_id, movie\_id),*

*FOREIGN KEY(director\_id) REFERENCES directors(id),*

*FOREIGN KEY(movie\_id) REFERENCES movies(id)*

*);*

Вольный перевод SQL-запроса на русский язык:

* создаём таблицы **movies** и **directors** и промежуточную таблицу **directors\_movies**;
* в промежуточной таблице указываем связь между записями из двух таблиц;
* в таблице **directors\_movies** создаём композитный первичный ключ из полей director\_id и movie\_id; при этом решается и вторая задача: теперь каждая пара полей будет уникальна в пределах таблицы и никто не сможет повторно связать фильм с режиссёром.

Теперь любую запись из таблицы **directors** можно связать с несколькими фильмами из **movies**, а фильм можно связать с несколькими режиссёрами.

1. Задача

К мороженому может прилагаться *топпинг* — какая-то добавка, которая делает мороженое ещё вкуснее (даже если вкуснее некуда).

Есть множество топпингов, их названия хранятся в таблице toppings.

Свяжите таблицы ice\_cream и toppings через промежуточную таблицу так, чтобы у мороженого могло быть несколько топпингов, а могло бы не быть вовсе. У разных сортов мороженого могут быть одинаковые топпинги.

В промежуточной таблице обязательно создайте колонку с первичным ключом.

Ключ в промежуточной таблице, связывающей ice\_cream и toppings, **не должен быть композитным**. Связь мороженного и топпинга — это неуникальная комбинация: к одному мороженому можно привязать не один, а два, пять или десять одинаковых топпингов.

Поэтому в таблице обязательно нужна колонка id PRIMARY KEY.

Создайте промежуточную таблицу ice\_cream\_toppings с полями

* id: первичный ключ, INTEGER;
* ice\_cream\_id: обязательное поле, INTEGER;
* topping\_id: обязательное поле, INTEGER;

Поля ice\_cream\_id и topping\_id должны быть внешними ключами и ссылаться «один-ко-многим» на таблицы ice\_cream и toppings.

Код:

import sqlite3

con = sqlite3.connect('db.sqlite')

cur = con.cursor()

cur.executescript('''

CREATE TABLE IF NOT EXISTS toppings(

id INTEGER PRIMARY KEY,

name TEXT NOT NULL

);

CREATE TABLE IF NOT EXISTS ice\_cream(

id INTEGER PRIMARY KEY,

name TEXT NOT NULL

);

CREATE TABLE IF NOT EXISTS ice\_cream\_toppings(

id INTEGER PRIMARY KEY,

ice\_cream\_id INTEGER NOT NULL,

topping\_id INTEGER NOT NULL,

FOREIGN KEY(ice\_cream\_id) REFERENCES ice\_cream(id),

FOREIGN KEY(topping\_id) REFERENCES toppings(id)

);

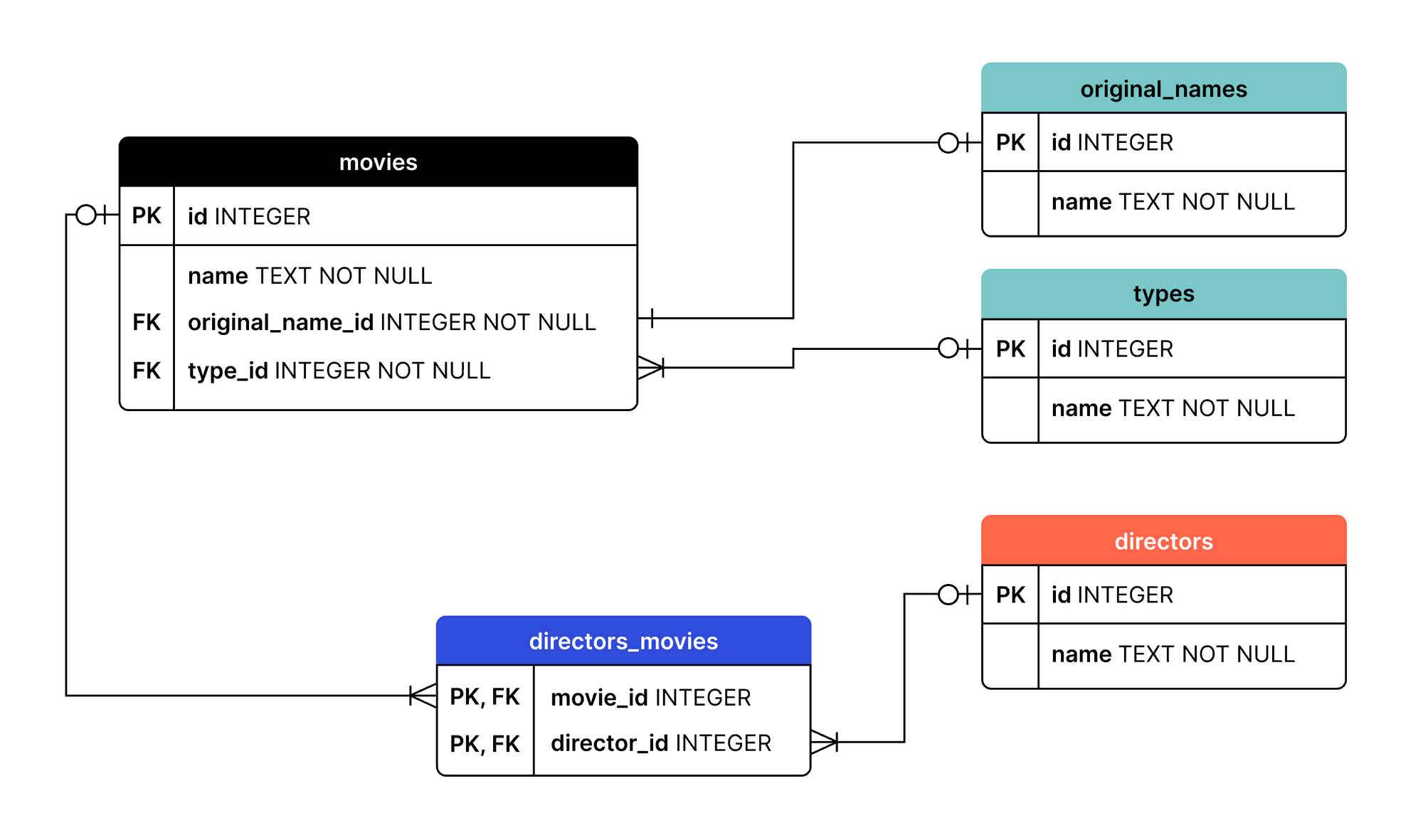
''')

con.commit()

con.close()

**16/17 Изменение таблиц в БД. Ссылочная целостность**

На основе всех проведённых экспериментов можно собрать базу данных из четырёх связанных таблиц.



Если база данных спроектирована идеально и в её архитектуре учтены все возможные и невозможные варианты развития проекта — можно создать таблицы и больше к ним не возвращаться.

Однако в реальном мире база данных — это живая структура, она может расти и изменяться вместе с проектом.

Ведь в ходе разработки или эксплуатации проекта:

* может расшириться или измениться техническое задание;
* могут появиться новые данные, ради которых придётся изменить структуру БД;
* да и вообще — всякое бывает.

В любой из перечисленных ситуаций потребуется отредактировать таблицы в БД:

* переименовать, добавить или удалить таблицы,
* переименовать, добавить или удалить столбцы в таблицах.

### Изменение таблицы: ALTER TABLE

Структуру таблиц можно изменять: можно добавлять, переименовывать или удалять колонки; можно переименовывать или удалять таблицы. Это делается командой ALTER TABLE имя\_таблицы описание\_операции.

**Переименование таблицы:**

*ALTER TABLE <имя таблицы> RENAME TO <новое имя таблицы>;*

**Добавление колонки:**

*ALTER TABLE <название таблицы>*

*ADD COLUMN <имя колонки> <тип колонки>;*

По умолчанию SQLite добавляет новую колонку в конец списка колонок.

**Переименование колонки:**

*ALTER TABLE <название таблицы>*

*RENAME COLUMN <старое имя колонки> TO <новое имя колонки>;*

**Удаление колонки:**

В классическом SQL сработает вот такая конструкция:

*ALTER TABLE <название таблицы>*

*DROP COLUMN <имя колонки>;*

Но в SQLite удаление колонок доступно только с версии 3.35.0. Для более старых версий для удаления колонки приходится идти сложным путём:

1. Создать копию таблицы, но без той колонки, которую нужно удалить.
2. Перенести в новую таблицу информацию из исходной таблицы.
3. Удалить исходную таблицу .
4. Переименовать созданную.

Документация SQLite [даст более детальную и глубокую информацию по работе с колонками](https://www.sqlite.org/lang_altertable.html).

### Удаление таблиц

"Aliena nobis, nostra aliis" (лат. «Чужое — нам, а наше — другим (нравится)»)

Сенека

«Ежели один человек построил, другой завсегда разобрать сможет»

Сенека в переводе кузнеца Степана

Команда для удаления таблицы выглядит так:

*DROP TABLE <имя таблицы>;*

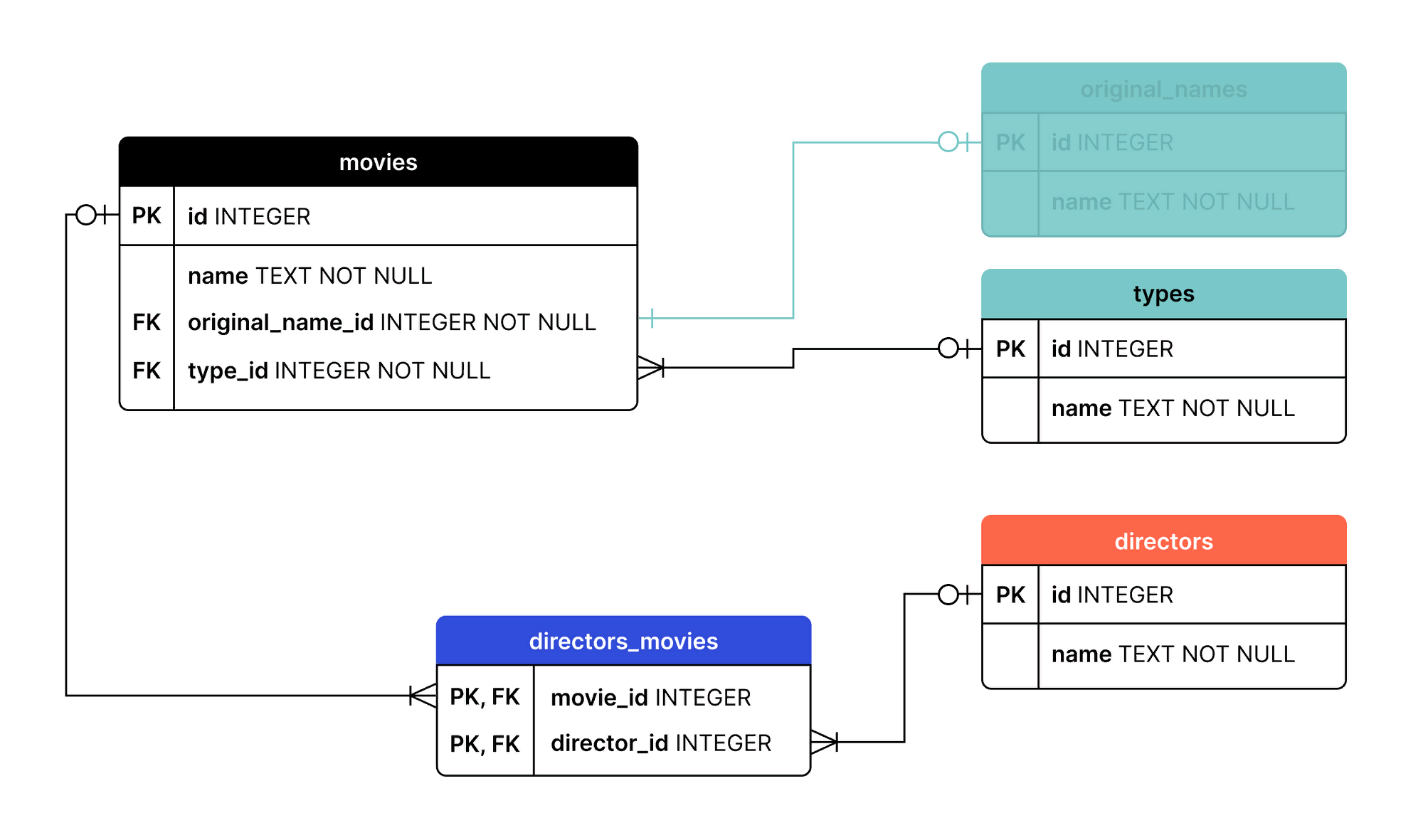
## Ссылочная целостность данных

При удалении таблиц есть серьёзная опасность: помимо того, что будут удалены данные, может быть нарушена **ссылочная целостность** базы. На удаляемую таблицу могут ссылаться другие таблицы; ссылки останутся, а таблицы не станет. База перестанет работать.

Удалим, для эксперимента, таблицу **original\_names**:

*DROP TABLE original\_names;*

Ничего хорошего не получится: ссылка из колонки original\_name\_id таблицы **movies** повиснет в воздухе.



Подобная же история может случиться и при удалении колонок и при их переименовании.

При изменении первичных и внешних ключей важно соблюдать **ссылочную целостность данных** (referential integrity). Основная идея в том, чтобы не было «битых», ведущих в никуда, ссылок на другие таблицы.

Вот примеры нарушений целостности данных:

* **Аномалия удаления** (*deletion anomaly*): если из таблицы удалена строка, на которую ссылается внешний ключ.
* **Аномалия вставки** (*insertion anomaly*): если добавлена запись с внешним ключом, но этот внешний ключ не соответствует ни одному первичному ключу из связанной таблицы.
* **Аномалии обновления** (*update anomaly*): при изменении данных в одной строке они могут прийти в противоречие с данными из другой строки: например, изменён PK, на который ссылается FK из другой таблицы.

1. Задача

В веб-проекте администратор может снимать с публикации какие-то записи, не удаляя их из БД. За это в нашем проекте будет отвечать поле is\_published в таблице ice\_cream. А чтобы отметить, какие сорта мороженого будут показаны на главной странице проекта, а какие нет — понадобится поле is\_on\_main.

В таблицу ice\_cream добавьте обязательные колонки is\_published и is\_on\_main булева типа.

Для добавления двух колонок используйте ALTER TABLE ... ADD COLUMN ...; ALTER TABLE ... ADD COLUMN ...;.

В SQLite нет отдельного булева типа. Для описания столбца используется тип INTEGER и булевы значения хранятся как 0 (False) и 1 (True).

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.executescript('''*

*ALTER TABLE ice\_cream*

*ADD COLUMN is\_published INTEGER;*

*ALTER TABLE ice\_cream*

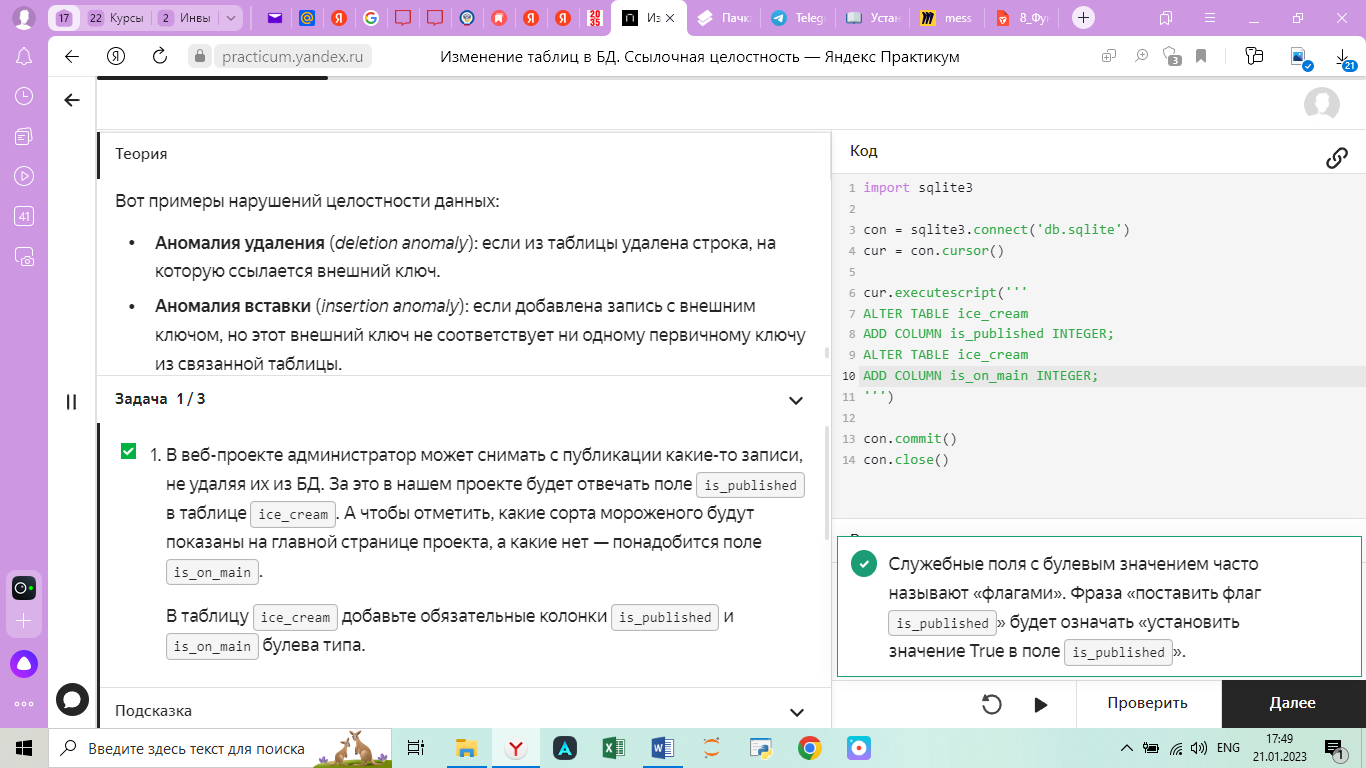
*ADD COLUMN is\_on\_main INTEGER;*

*''')*

*con.commit()*

*con.close()*

Результат:



2. Задача

В таблице ice\_cream переименуйте колонку description. Новое название должно быть specification.

Используйте RENAME COLUMN.

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.executescript('''*

*ALTER TABLE ice\_cream*

*RENAME COLUMN description TO specification;*

*''')*

*con.commit()*

*con.close()*

3. Задача

Удалите таблицу ice\_cream из БД.

Используйте DROP TABLE .

Код:

*import sqlite3*

*con = sqlite3.connect('db.sqlite')*

*cur = con.cursor()*

*cur.executescript('''*

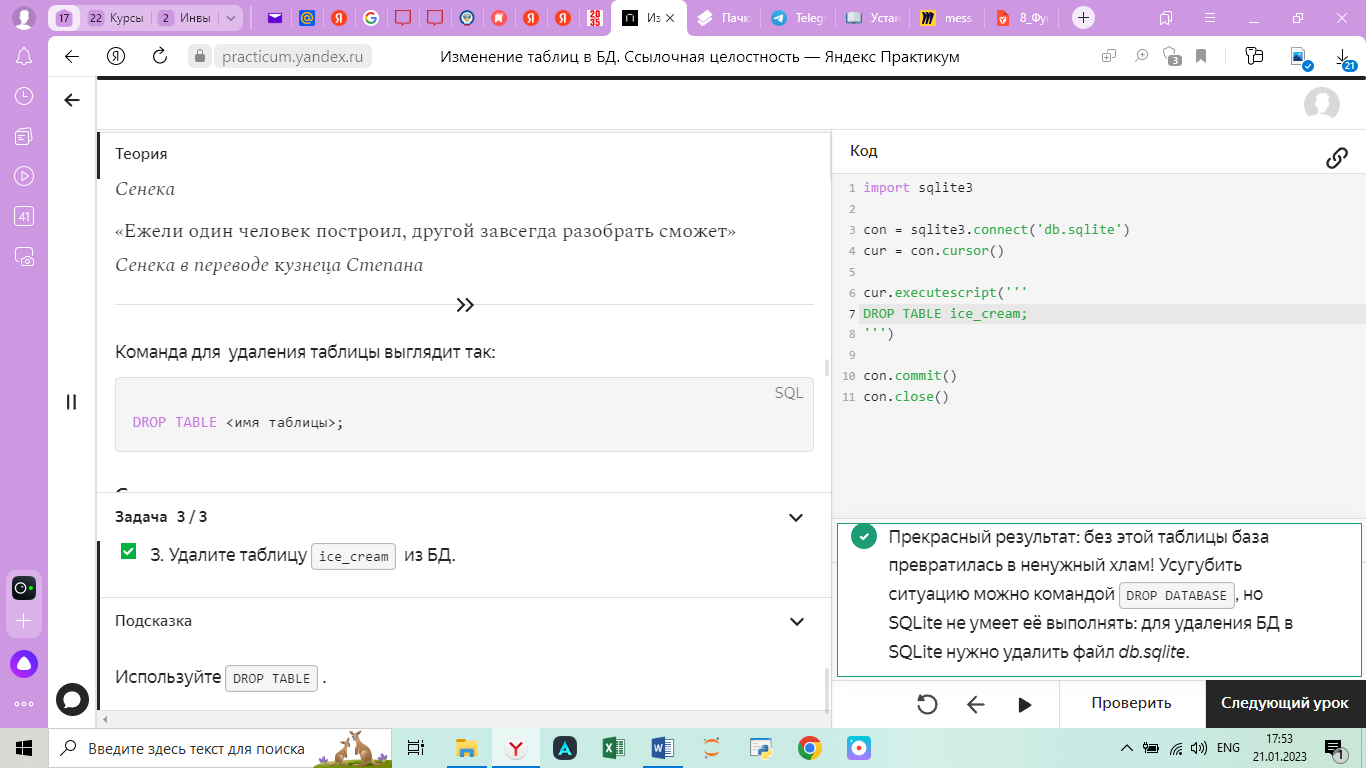
*DROP TABLE ice\_cream;*

*''')*

*con.commit()*

*con.close()*

Результат:



# Тема 4/4: Команда единомышленников

# Студенческое коммьюнити: вместе — легче

Обычно в школе или в университете студенты объединяются в группы, общаются на переменах, собираются после лекций, вместе готовятся к экзаменам, да и во время лекций много чего успевают обсудить (особенно на задних рядах). В этом — взаимная поддержка и радость общения с хорошими людьми.

При онлайн-обучении непосредственного общения бывает меньше, но этот недостаток можно и нужно компенсировать, выстраивая **коммьюнити** — сообщество людей, объединённых общими интересами и целями.

На нашем курсе у всех участников обучения есть общая цель: превратить студентов в бэкенд-разработчиков. Эта цель объединяет и студентов, и наставников, и кураторов — это значимый мотив для создания сообщества.

### Кто поймёт студента? Другой студент!

В Практикуме вы не останетесь один на один с платформой: вас всегда поддержат наставники и кураторы, а ближе к концу обучения присоединятся и менеджеры по трудоустройству.

Однако главный секрет успешного обучения — активное общение с однокурсниками. Учиться будет гораздо легче, если есть с кем обсудить проблемы и поделиться открытиями.

Столкнувшись со сложностями, студент начинает переживать: «Кажется, я здесь хуже всех». И только в общении с сокурсниками выясняется, что сложно — всем. Как ни странно — от этого становится легче: «проблема не во мне!».

Коммьюнити Практикума — это безопасная среда, где вы можете смело общаться с другими разработчиками, обсуждая сложные темы.

### Фундамент коммьюнити

Основная площадка, на которой вы будете знакомиться и общаться с другими студентами — это мессенджер Пачка. Чем быстрее вы в нём освоитесь, тем легче и интереснее будет учиться.

Вот несколько вариантов, которые помогут ближе познакомиться с однокурсниками:

* Расскажите о себе в канале **#offtopic**. Прочитайте посты других студентов, прокомментируйте их.
* Следите за вопросами в каналах **#theory** и **#project**, при возможности помогайте другим студентам.
* Возможно, у вас уже есть вопросы; задайте их в одном из каналов.
* Расскажите в канале **#offtopic**, как у вас прошли первые недели обучения. Если есть сложности и страхи, поделитесь ими: вот увидите, вы не одиноки.
* Предложите однокурсникам неформальный созвон и оффлайн встречу.
* Чтобы узнать друг друга получше — в канале **#offtopic** задайте вопрос однокурсникам или наставнику, попросите их рассказать что-нибудь интересное или забавное.

# Поиск решения

— Ура, у меня получилось! Я же Альберт Эйнштейн и Кэтрин Джонсон одновременно!

— Всё пропало, я ни на что не гожусь, и зачем было ввязываться в это дело...

Эти два состояния будут регулярно сменять друг друга во время обучения, и если первый вариант имеет под собой серьёзные основания, то второй — лишь следствие усталости или непонимания, как подступиться к проблеме.

Любая проблема — это просто задача, которая требует решения. А практические шаги для решения такой задачи можно свести к трём основным этапам:

1. Сформулировать, в чём же состоит проблема («ничего не получается» звучит слишком расплывчато).
2. Поискать решение самостоятельно.
3. Если решение не нашлось — обратиться за помощью к наставникам и однокурсникам.

## В чём проблема?

Студенты Практикума чаще всего сталкиваются с проблемами в трёх категориях:

* **Проблемы в теории:** незнакомые термины, сложный материал в теоретической части курса, материал вебинара похож на китайскую грамоту.
* **Проблемы в практике:** непонятно, что и как нужно сделать в задании, программа выдаёт ошибки; действия по инструкции не приводят к ожидаемому результату.
* **Организационные проблемы:** демотивация, форс-мажоры, непонимание учебного процесса.

Определите, к какой категории относится проблема, сформулируйте её и определите вопросы, ответы на которые нужно найти.

Если вам непонятен конкретный раздел в теории, разбейте проблему на мелкие фрагменты. Возможно, у вас не один вопрос, а несколько; а может и наоборот: вы разобрались во всём, кроме одного небольшого вопроса.

Если сложности возникли в практической части — отмотайте плёнку назад, восстановите цепочку событий: в какой момент и после каких действий всё пошло не так. Сформулируйте вопрос: в чём конкретно нужна помощь.

Точно так же можно поступить с организационными вопросами: если словами описать проблему — можно будет сформулировать конкретный вопрос, с решением которого требуется помощь в первую очередь..

## Самостоятельный поиск ответа

Не спешите задавать вопросы сокурсникам или наставнику, вначале исследуйте вопрос самостоятельно.

### Возможно, ответ есть в уроках

Когда вопрос сформулирован — ответ может найтись в пройденных уроках, в теории или в практических заданиях.

* Перечитайте задание и теорию, и формулировки станут понятнее. В теории есть ответы и на организационные вопросы — во вводных уроках в начале курса.

### Возможно, ваша проблема не уникальна

Есть шанс, что с проблемой, которую вы пытаетесь решить, кто-то из ваших однокурсников уже столкнулся — а может, и решил её.

* Проверьте каналы в мессенджере, поищите по нескольким вариантам запросов: возможно, эту проблему уже обсуждали и нашли решение.

### Поиск в интернете

Даже если никто из ваших однокурсников не сталкивался с проблемой, подобной вашей — будьте уверены: в мировой практике этот вопрос уже поднимался, и на него уже найден ответ, если не в русском сегменте интернета — то уж точно в английском. Вперёд, в глобальную сеть!

**Алгоритм поиска в интернете**

Этот алгоритм опирается на отличие: как люди задают вопрос собеседнику и как — поисковику. В первом случае формулировку сначала обдумывают, а потом озвучивают, чтобы не тратить время и нервы друг друга. С поисковиком же работают, пока не будут удовлетворены ответом или просто не надоест. Поэтому можно, долго не думая, ввести вопрос в поисковую строку, а потом его улучшить.

**Правильный запрос — половина успеха**

Подумайте, что и в какой терминологии может быть написано на той странице, которая хранит ответ на ваш вопрос. Ищите слова с этой воображаемой страницы.

Скорее всего, после первого-второго запроса поиск не даст нужного ответа. Но обзор поисковой выдачи поможет понять, как искать дальше.

* Измените слова в запросе на синонимы
* Воспользуйтесь синтаксисом расширенного поиска (инструкции для [Яндекса](https://yandex.ru/support/search/query-language/search-context.html); для [Google](https://support.google.com/websearch/answer/2466433?hl=ru));
* Уточните запрос, исходя из найденных результатов.
* Уберите из запроса лишние слова.
* Разбейте вопрос на части, чтобы уточнить детали.
* Перейдите на англоязычные запросы — в этом случае вероятность найти ответ возрастёт многократно.
* Ищите в нескольких поисковых системах.

### Поиск в документации

У каждого языка программирования, у каждой серьёзной библиотеки или модуля есть официальное описание, созданное авторами, — документация. Обычно документация есть в открытом доступе.

Если вы столкнулись с ошибкой в коде и не понимаете, в чём дело, — читайте документацию.

Найти документацию несложно: достаточно, например, сделать поисковый запрос по словам Python документация. Такой запрос приведёт к сайту, где размещён перевод официальной документации на русский язык. При чтении документации обращайте внимание на версию языка или библиотеки, чтобы не погрузиться в изучение устаревшего материала.

Если результаты по простому запросу вас не удовлетворят, уточните его — например, Python функция len() документация. В ответ поисковик выдаст [ссылку на официальную документацию языка](https://docs.python.org/3/library/functions.html).

Документация не всегда бывает удобна, красива и понятна, но даже если вы не увидите в ней прямого ответа на свой вопрос — вы найдёте там подсказки, что и как искать дальше.

Удачной охоты!

Поиск может и не дать ответа на ваш вопрос, но при любом исходе вы получите дополнительную информацию по интересующей вас теме. В таком случае — вперёд в комьюнити, задавать вопросы!

# Зачем разработчику задавать вопросы

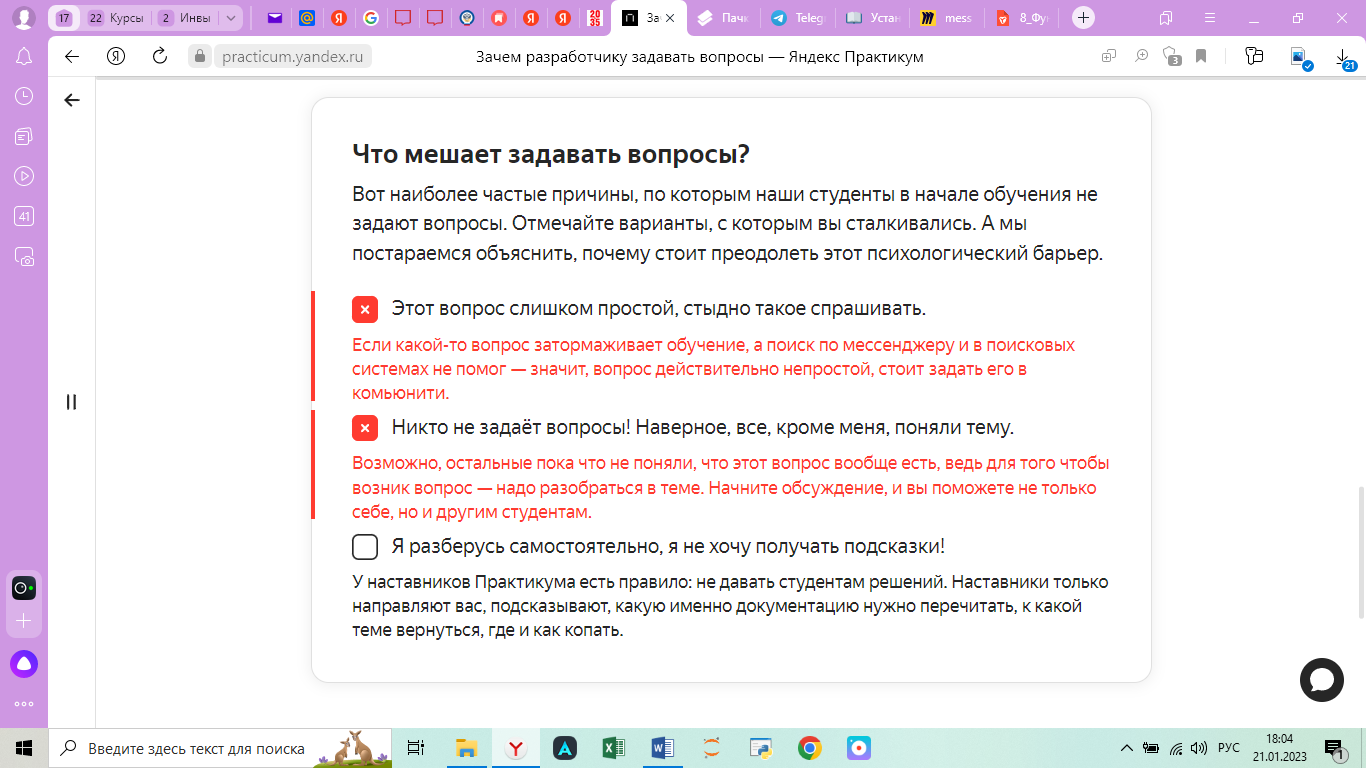
Разработка — это постоянная коммуникация со всеми участниками команды: тимлидами, продакт- и проджект-менеджерами, тестировщиками, другими программистами. Это постоянный обмен знаниями и опытом, взаимодействие по возникающим вопросам и координация действий (хотя приветствия по утрам и совместные походы на обед — это тоже здорово).

Вопросы задают все, даже ведущие разработчики с большим опытом; что же касается начинающих программистов — чем больше вопросов они задают, тем быстрее набираются опыта.

## Зачем задавать вопросы?

* Задавая грамотные вопросы в каналах Практикума, вы получите больше, чем даёт платформа. Глубокие знания и навыки приобретаются не при чтении учебников, а при самостоятельном поиске. Именно поэтому наставники Практикума не дают готовых ответов, а направляют студентов к самостоятельным решениям.
* Работодатель по вашим вопросам (на собеседовании и во время испытательного срока) будет оценивать ваш ход мысли, умение анализировать, подход к обработке задач. Ваши вопросы — это второе портфолио, способ продемонстрировать себя и свои навыки, подход к решению задач и проблем
* Вопросы помогают сэкономить время. Да, перед тем как задавать вопросы, важно уделить время самостоятельному поиску. Но здесь важно не увлечься и не потратить весь день на то, чтобы найти ответ на один вопрос. Если через пару часов решение не нашлось — самое время задать вопрос.
* Индустрия быстро растёт и меняется, за ней не успеть. На протяжении всей карьеры вам придётся задавать вопросы тем, кто в курсе новых технологий. Огромную часть вашего времени будет занимать общение с коллегами, иначе не бывает.

Опытные разработчики задают хорошо структурированные вопросы, без лишних слов и с достаточным количеством исходных данных. Хороший вопрос во многом похож на красивый код. Вы научитесь этому в ближайших уроках и на практике, это выгодно отличит вас от других джунов-разработчиков на собеседованиях и в работе.



Многие студенты не решаются задавать вопросы, но этот навык приходит с опытом. Понаблюдайте за собой в течение следующей недели: получается ли у вас задавать вопросы в чате, когда вам действительно нужна помощь? Если нет — какие мысли мешают вам это делать? Спрашивать и искать ответы — это точно такой же профессиональный навык, как умение писать код, и этот навык требует практики.

**Как задавать вопросы**

В предыдущих уроках вы прочитали, почему важно задавать вопросы во время обучения, даже если вам страшно и неловко это делать. Но вопросы бывают разные. На одни ваши коллеги будут отвечать быстро и подробно, на другие вы будете получать много ответных уточняющих сообщений — и обсуждение проблемы затянется надолго. В этом уроке мы расскажем, как составлять первый вариант вопросов — их называют содержательными.

Содержательный вопрос показывает тому, кто его прочитал, что вы столкнулись с проблемой, пробовали решать её самостоятельно, подумали — и только потом пришли за помощью. Они выстроены так, что адресат быстро понимает, в чём проблема: ему удобно читать и отвечать на такой вопрос.

## Алгоритм составления содержательного вопроса

В любых сообщениях не стоит писать просто «Привет!» и ждать, пока собеседник ответит: лучше сразу написать свою просьбу или вопрос в форме лаконичного сообщения.

Вопрос, который вы задаёте коллегам, должен содержать вводную информацию:

* что вы планировали сделать и что сделали;
* какая проблема возникла, в чём она заключается или в чём выражается;
* ваши предположения: почему возникла проблема и что может помочь её решить
* как вы пробовали решить проблему;
* в чём конкретно нужна помощь.

На все эти вопросы можно ответить устно, а можно подробно расписать детали в черновике. Зачастую ответ приходит в тот момент, когда вопрос сформулирован в письменном виде.

Если решение не пришло, то нужно лаконично и грамотно оформить все ваши мысли по проблеме.

## Как правильно оформить вопрос в мессенджере

**Договоримся о правилах работы в каналах:**

* Выберите канал, тематически подходящий для вашего вопроса:
  + **#theory** — для вопросов по теории,
  + **#project** — по проекту в финале спринта,
  + **#offtopic** — для организационных и около учебных вопросов.
* Начните с заголовка. Формат: Номер спринта. Тема. Урок. Номер задачи.   
  Выделяйте заголовок.
* Коротко опишите, что делали, какая проблема возникла.
* В одном предложении сформулируйте вопрос, на который хотите получить ответ. Если вопросов несколько — оформите их в виде списка.
* Дополнительные материалы (ваш код, код или текст ошибки, скриншот приложения) прикрепляйте в комментарии к посту.
* Код оформите через функцию text snippet.
* Если ответ на вопрос найден — отметьте пост галочкой (есть такой эмоджи), это поможет другим студентам понять статус обсуждения, не читая 100500 комментариев.

## На мой вопрос не отвечают. Что делать?

Если на ваш вопрос не ответили — обязательно напоминайте о нём. Если вопрос задан наставнику или куратору — отмечайте их в комментах через символ @. А если вам ответили, но вы не поняли ответ — не стесняйтесь переспросить, уточните вопрос, попросите объяснить более доступно.

**Если вам не отвечают на вопрос в чатах Практикума:**

* Если прошло 24 часа, а ответа нет — пришлите куратору ссылку на ваш вопрос в чате.
* Если ответ наставника не помог вам разобраться с вопросом — не стесняйтесь переформулировать вопрос или позвать на помощь куратора.
* Если на ваш вопрос ответили одногруппники и не успел ответить наставники — это тоже неплохо: наставники просматривают все обсуждения и корректируют ответы там, где это нужно.

Если же вы разобрались с теорией так, что вопросов не осталось — подумайте, о чём вам было бы интересно спросить своего наставника. Составьте грамотный и интересный вопрос, опубликуйте его в канале #**offtopic**.