



Методы сбора и обработки данных при помощи Python

Системы управления базами данных MongoDB и SQLite   
в Python

**На этом уроке**

1. Разберём основные принципы работы с реляционными и нереляционными базами данных.
2. Рассмотрим основные операции и методы для формирования запросов.
3. Научимся работать с данными внутри баз.

**Оглавление**

[SQL и noSQL](#_n707dgkmyr8p)

[SQL](#_4d34og8)

[NoSQL](#_2s8eyo1)

[MongoDB и почему именно она](#_17dp8vu)

[Документная модель данных](#_3rdcrjn)

[Произвольные запросы](#_26in1rg)

[Работа с MongoDB](#_lnxbz9)

[Установка MongoDB](#_35nkun2)

[Запуск сервера на Windows](#_1ksv4uv)

[Содержимое пакета MongoDB](#_o8j8jf85rojj)

[Создание каталога для БД и запуск MongoDB](#_ac41blklr3ip)

[Запуск на MacOS](#_z7rltuqxdq1p)

[Работа с MongoDB из консоли](#_3ygqlhmzd3w8)

[Операции CRUD](#_44sinio)

[Работа с MongoDB в Python](#_2jxsxqh)

[Работа с SQLite](#_yulmtcjuu4ge)

[Установка и подключение к БД](#_z337ya)

[Домашнее задание](#_8ikobe3t4yv7)

[Глоссарий](#_eysn9k2ji11s)

[Дополнительные материалы](#_3j2qqm3)

*При написании руководства были использованы ОС: Windows 10 x64, Python v3.7.2, PyCharm Edu v2019.3.2, MongoDB v4.2.0*

# SQL и noSQL

СУБД (DBMS, Database Management System) — система управления базами данных. Это набор команд, прикладных и инфраструктурных приложений, а также библиотек, которые позволяют управлять и обслуживать базу данных. По сути, как API для программиста.

База данных (БД) — структурированный массив данных. Он может располагаться как на жёстком диске, так и в оперативной памяти. Есть множество типов СУБД, но сейчас актуальны два — реляционные (SQL) и нереляционные (noSQL).

## SQL

Реляционные БД основаны на реляционной модели данных. Базы данных SQL используют язык структурированных запросов для определения данных и управления ими. SQL — универсальный и широко используемый вариант, который хорошо подходит для сложных запросов и гарантирует безопасность. Однако он может носить ограничительный характер, поскольку требует использования предопределённых схем для определения структуры данных перед работой с ними. У всех данных должна быть одинаковая структура, так что предварительная подготовка может отнять немало усилий.

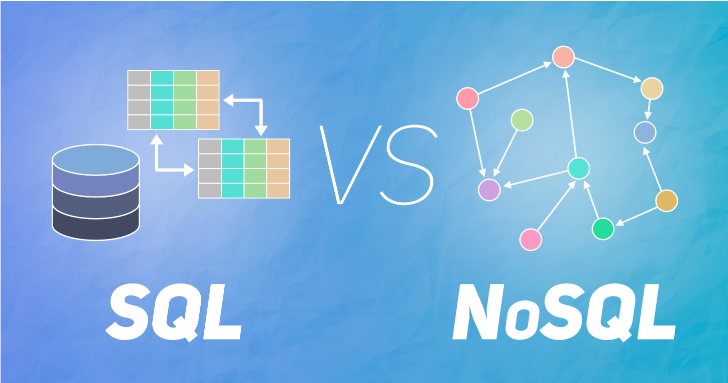
## NoSQL

Базы данных NoSQL, наоборот, имеют динамические схемы для неструктурированных данных. Данные здесь хранятся разными способами: они могут быть ориентированы на столбцы или на документы, основаны на графах или организованы как хранилище KeyValue. Такая гибкость означает, что:

* вы можете создавать документы без предварительного определения их структуры;
* каждый документ может иметь свою уникальную структуру;
* синтаксис может варьироваться от базы данных к базе данных;
* вы можете в любой момент добавлять новые поля.

Всего есть 4 типа баз данных NoSQL:

* **Ключ-значение** (Redis, Berkeley DB). Хранилища типа key-value — простейшие БД, чаще всего это in-memory базы данных (то есть они хранятся и работают в оперативной памяти сервера). Такой БД не требуется ни схемы, ни связи.
* **Документоориентированные** (MongoDB, CouchDB). В таких БД данные хранятся в виде документов и имеют сложную иерархическую (древовидную) структуру, между элементами есть связи.
* **Графовые** (Giraph, Neo4j). Такие БД позволяют хорошо реализовать семантические паутины (например, социальные сети), в подобных задачах они более производительны;



* **BigTable** (HBase, Cassandra). В них данные представлены в виде разреженной матрицы. Эти БД похожи на документоориентированные.

# MongoDB и почему именно она

MongoDB — документоориентированная СУБД с открытым исходным кодом. Не требует схемы таблиц и относится к NoSQL. Хранит данные в виде документов в формате JSON.

Преимущества MongoDB:

* скорость разработки;
* нет необходимости в поддержке схемы и в коде, и в БД;
* лёгкая масштабируемость;
* гибкость при смене задачи;
* удобство работы с денормализованными данными.

Почему MongoDB подходит под задачу сбора и анализа данных:

* данные быстро меняются (дополнительные данные из API, динамический контент в HTML-страницах);
* БД нужна лишь до тех пор, пока нужны данные;
* данные постоянно обновляются;
* нормализация не нужна;
* задача не меняется.

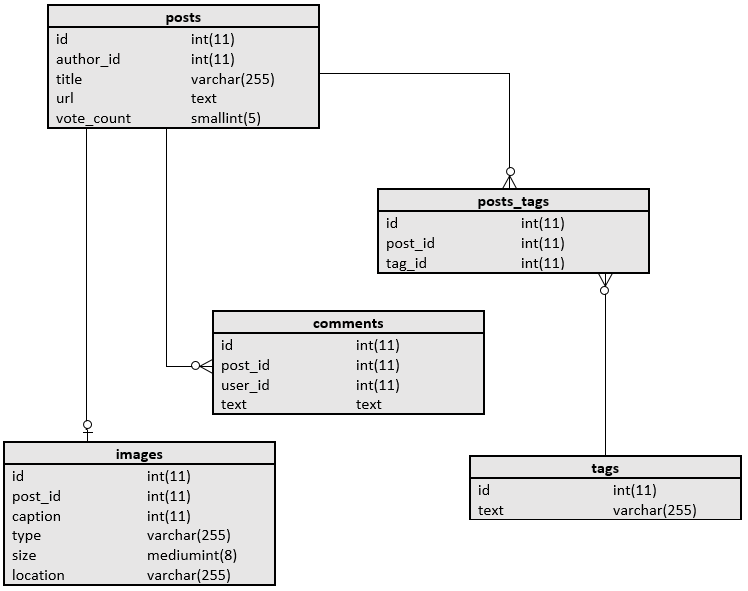
## Документная модель данных

Модель данных MongDB — документоориентированная. Для тех, кто не знаком с идеей документа в контексте баз данных, продемонстрировать её проще всего на примере.

| {\_id: ObjectID("4bd9e8e17cefd644108961bb"), *#Поле \_id - первичный ключ*  title: "Adventure in Databases",  url: "http://example.com/databases.txt",  author: "msmith",  vote\_count: 20,   tags: ['databases', 'mongodb', 'indexing'], *#Теги хранятся в виде массива строк*   image:{ *#Атрибут указывает на другой элемент*  url: "http://example.com/db.jpg",  caption:"",  type: "jpg",  size: 75381,  data: "Binary" result.append(result\_v) },  comments: [ *#Комментарии хранятся в виде массива объектов,*   { *#представляющих ещё один комментарий*  user: "bjones",  text: "Interesting article!"  },  {  user: "blogger",  text: "Another related article is at http://example.com/db/db.txt"  }  ] } |
| --- |

Это пример документа, представляющего статью на социальном новостном сайте. Как видите, документ — это набор, состоящий из имён и значений свойств. Значение может быть представлено простым типом: например, строки, числа и даты. Но может быть также последовательностью и даже другим документом. С помощью таких конструкций можно представлять весьма сложные структуры данных. Так, в нашем примере имеется свойство *tags* — список, в котором хранятся ассоциированные со статьёй теги. Но ещё интереснее свойство *comments*, которое ссылается на список документов, содержащих комментарии.

Сравним это с представлением тех же данных в стандартной реляционной базе:



Здесь изображён типичный реляционный аналог. Поскольку таблицы, по сути, плоские, для представления связей типа «один-ко-многим» нужно несколько таблиц. Мы начинаем с таблицы posts, в которой хранится основная информация о каждой статье. Затем создаём ещё три таблицы, каждая из которых содержит поле *post\_id*, ссылающееся на исходную статью.

Вероятно, вы обратили внимание, что документы не только позволяют представлять данные со сложной структурой, но и не нуждаются в заранее определённой схеме. В реляционной базе данных строки хранятся в таблице. У каждой таблицы строго определённая схема, описывающая, какие столбцы и типы данных допустимы. Если окажется, что нужно добавить ещё одно поле, таблицу придётся менять.

В MongoDB документы группируются в коллекции — контейнеры, не налагающие на данные какую-либо схему. Теоретически у каждого входящего в коллекцию документа может быть своя структура, но на практике документы в одной коллекции похожи друг на друга. Например, у всех документов в коллекции posts есть поля title, tags, comments и так далее.

## Произвольные запросы

Рассмотрим принцип построения запросов в MongoDB на простом примере статей и комментариев. Пусть нужно найти все статьи, помеченные тегом politics, за которые проголосовало более 10 посетителей. SQL-запрос для решения этой задачи выглядел бы так:

| SELECT \* FROM posts INNER JOIN posts\_tags ON posts.id = posts\_tags.post\_id INNER JOIN tags ON posts\_tags.tag\_id == tags.id WHERE tags.text = 'politics' AND posts.vote\_count > 10; |
| --- |

Эквивалентный запрос в MongoDB формулируется путём задания документа-образца. Условие «больше» обозначается специальным ключом *$gt*.

| db.posts.find(('tags': 'politics', 'vote\_count': {'$gt': 10}}); |
| --- |

Стоит отметить, что в этих запросах предполагаются разные модели данных. SQL-запрос опирается на строго нормализованную модель, в которой статьи и теги хранятся в разных таблицах, поэтому мы объединяем таблицы при помощи *JOIN*, тогда как в запросе для MongoDB считается, что теги хранятся внутри документа, описывающего статью.

# Работа с MongoDB

## Установка MongoDB

MongoDB — кроссплатформенная СУБД. Чтобы скачать дистрибутив, нужно перейти по ссылке [Download Center: Community Server](https://www.mongodb.com/download-center/community), выбрать свою ОС и нажать кнопку Download. А затем установить скачанный архив, следуя подсказкам мастера установки.

Инструкция по установке — [Install MongoDB Community Edition](https://docs.mongodb.com/manual/administration/install-community/).

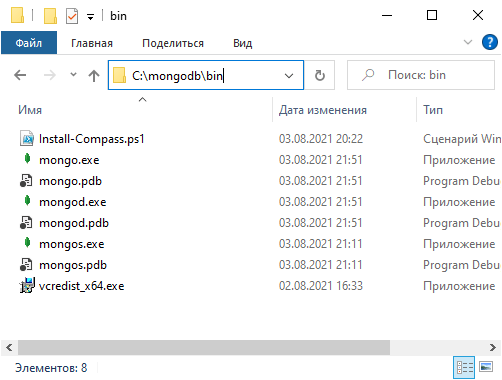
## Запуск сервера на Windows

После загрузки архивного пакета распакуем его в папку **C:\mongodb**.

Подробнее — [Установка и начало работы с MongoDB на Windows](https://metanit.com/nosql/mongodb/1.2.php).

## Содержимое пакета MongoDB

Если после установки мы откроем папку **bin** в распакованном архиве (*C:\mongodb\bin*), то сможем найти там кучу приложений, которые выполняют определённую роль. Вкратце рассмотрим их.



* **mongo** — консольный интерфейс для взаимодействия с базами данных, своего рода консольный клиент;
* **mongod** — сервер баз данных MongoDB, обрабатывает запросы, управляет форматом данных и выполняет различные операции в фоновом режиме по управлению базами данных;
* **mongos** — служба маршрутизации MongoDB, которая помогает обрабатывать запросы и определять местоположение данных в кластере MongoDB.

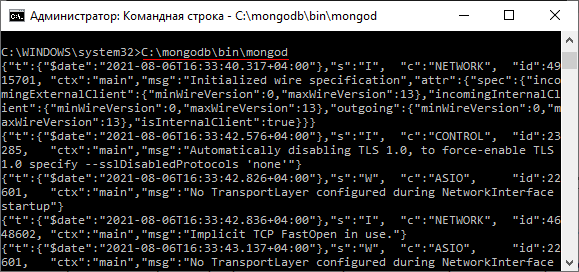
## Создание каталога для БД и запуск MongoDB

После установки надо создать на жёстком диске каталог, в котором будут находиться базы данных MongoDB.

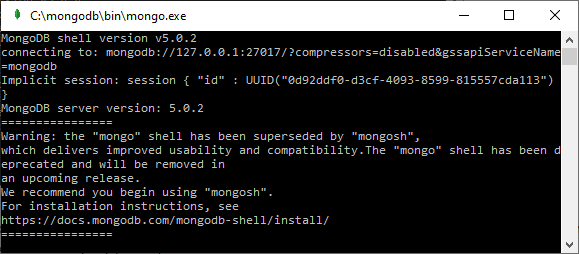
В ОС Windows по умолчанию MongoDB хранит базы данных по пути **C:\data\db**, поэтому, если вы используете Windows, вам надо создать соответствующий каталог.

Если возникла необходимость использовать какой-то другой путь к файлам, его можно передать при запуске MongoDB во флаге **--dbpath.**

После создания каталога для хранения БД можно запустить сервер MongoDB. Сервер представляет приложение **mongod**, которое находится в папке bin. Для этого запустим терминал/командную строку и там введём соответствующие команды. Для ОС Windows это будет выглядеть так:



Командная строка отобразит нам ряд служебной информации, например, что сервер запускается на localhost на порту 27017. После удачного запуска сервера мы сможем производить операции с БД через оболочку **mongo**. Эта оболочка представляет файл **mongo.exe**, который располагается в выше рассмотренной папке установки. Запустим этот файл:



Это консольная оболочка для взаимодействия с сервером, через которую можно управлять данными. Второй строкой эта оболочка говорит о подключении к серверу mongod.

## Запуск на MacOS

Для запуска сервера MongoDB выполним в терминале команду:

| brew services start mongodb-community@5.0 |
| --- |

Для отключения сервера — команду:

| brew services stop mongodb-community@5.0 |
| --- |

Итак, после удачного запуска сервера мы сможем производить операции с БД через оболочку mongosh. Запустим её, выполнив в терминале команду **mongosh.**

## Работа с MongoDB из консоли

Чтобы узнать, какие базы данных у вас есть:

| > show dbs admin 0.000GB config 0.000GB local 0.000GB test 0.000GB |
| --- |

Чтобы подключиться к текущей базе данных или создать новую, используем команду *use*:

| > use newdb switched to db newdb |
| --- |

**Важно!** *В MongoDB используется принцип экономии: наша СУБД не будет создавать базу данных до тех пор, пока она пустая.*

Теперь произведём какие-либо простейшие действия. Введём в mongo последовательно следующие команды и после каждой нажмём enter:

| use mydb  db.users.insertOne( { name: "Tom" } )  db.users.find() |
| --- |

Первая команда **use test** устанавливает в качестве используемой базу данных mydb. Даже если такой БД нет, она создаётся автоматически. И далее db будет представлять текущую базу данных — то есть базу данных mydb. После db идёт users — это коллекция, в которую затем мы добавляем новый объект. Если в SQL нам надо создавать таблицы заранее, то коллекции MongoDB создаёт самостоятельно при их отсутствии.

С помощью метода **db.users.insertOne()** в коллекцию users базы данных mydb добавляется объект {name: "Tom" }. Описание добавляемого объекта определяется в формате, с которым вы, возможно, знакомы, если имели дело с форматом JSON. То есть в данном случае у объекта определён один ключ "name", которому сопоставляется значение "Tom". То есть мы добавляем пользователя с именем Tom.

Если объект был успешно добавлен, то консоль выведет результат операции, в частности, идентификатор добавленного объекта.

Третья команда **db.users.find()** выводит на экран все объекты из БД mydb.

Если в базе данных не остаётся ни одной коллекции, то она удаляется из общего списка. Удалим только что созданную коллекцию из БД:

| > db.users.drop() true |
| --- |

Проверим снова список существующих баз данных:

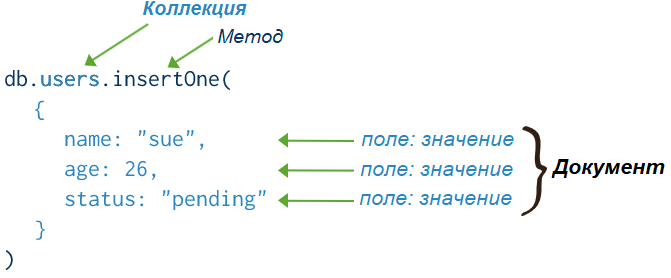
| > show dbs admin 0.000GB config 0.000GB local 0.000GB test 0.000GB |
| --- |

Как видим, база данных mydb также удалилась.

## Операции CRUD

CRUD-операции — это создание (Create), чтение (Read), обновление (Update), Удаление (Delete).

Общий синтаксис операции:



## Работа с MongoDB в Python

Прежде чем начать, необходимо установить модуль PyMongo:

| pip install pymongo |
| --- |

Первый шаг при работе с PyMongo — это подключение модуля MongoClient:

Затем создадим массив, который потом вставим в коллекцию

| cars = [ {'name': 'Audi', 'price': 52642},  {'name': 'Mercedes', 'price': 57127},  {'name': 'Skoda', 'price': 9000},  {'name': 'Volvo', 'price': 29000},  {'name': 'Bentley', 'price': 350000},  {'name': 'Citroen', 'price': 21000},  {'name': 'Hummer', 'price': 41400},  {'name': 'Volkswagen', 'price': 21600} ] |
| --- |

Создадим клиента для работы с монгой, передав ему имя хоста и порт:

| client = MongoClient('mongodb://localhost:27017/') |
| --- |

Посмотрим список баз данных:

| print(client.list\_database\_names()) |
| --- |

Создаём ссылку на базу данных testdb:

| db = client.testdb |
| --- |

С помощью метода *insеrt\_many()* передадим в коллекцию cars наш массив:

| db.cars.insert\_many(cars) |
| --- |

Существует также метод *insert\_one*, если вы хотите передать в коллекцию только один элемент из массива.

Теперь посмотрим все коллекции в базе данных с именем testdb:

| print(db.list\_collection\_names()) |
| --- |

**cars = db.cars.find()** — возвращает курсор, с помощью метода *next* мы можем идти по каждому элементу в переменной *cars*:

| print(cars.next()) |
| --- |

С помощью метода *list* мы можем трансформировать курсор в список и работать уже со списком:

| print(list(cars)) |
| --- |

Например, посчитать количество записей в базе данных:

| n\_cars = len(list(db.cars.find()))  print("There are {} cars».format(n\_cars)) |
| --- |

Также мы можем работать с элементами в переменной *cars* как со словарём:

| cars = db.cars.find()  for car in cars:  print('{0} {1}'.format(car['name'],  car[‘price'])) |
| --- |

Теперь найдём все машины, цена которых выше 50 000:

| expensive\_cars = db.cars.find({'price': {'$gt': 50000}})  for ecar in expensive\_cars:  print(ecar[‘name']) |
| --- |

В следующем примере выведем имя и id объектов коллекции:

| cars = db.cars.find({}, {'\_id': 1, 'name':1})  for car in cars:  print(car) |
| --- |

Отсортируем элементы по убыванию цены. Для этого импортируем *DESCENDING* и напишем следующее:

| cars = db.cars.find().sort("price", DESCENDING)  for car in cars:  print('{0} {1}'.format(car['name'],  car[‘price'])) |
| --- |

В примере вычисляется сумма всех цен на автомобили. Оператор *$sum* вычисляет и возвращает сумму числовых значений. Оператор *$group* группирует входные документы по указанному выражению-идентификатору и применяет выражения-аккумуляторы, если они указаны, к каждой группе.

| agr = [ {'$group': {'\_id': 1, 'all': { '$sum': '$price' } } } ] |
| --- |

Агрегатный метод применяет операцию агрегирования к коллекции автомобилей.

| val = list(db.cars.aggregate(agr))  print('The sum of prices is {}’.format(val[0]['all'])) |
| --- |

Давайте посчитаем сумму цен автомобилей Audi и Volvo:

| agr = [{ '$match': {'$or': [ { 'name': "Audi" }, { 'name': "Volvo" }] }},  { '$group': {'\_id': 1, 'sum2cars': { '$sum': "$price" } }}]  val = list(db.cars.aggregate(agr))  print('The sum of prices of two cars is {}’.format(val[0]['sum2cars'])) |
| --- |

Также мы можем пропускать определённое количество элементов таблицы с помощью метода *skip* и ограничивать количество выдаваемых результатов с помощью метода *limit*:

| cars = db.cars.find().skip(2).limit(3)  for car in cars:  print('{0}: {1}'.format(car['name'], car[‘price'])) |
| --- |

Удалить коллекцию cars из базы данных testdb:

| db.cars.drop() |
| --- |

Мы рассмотрели основные команды для работы с mongodb. Более подробную информацию ищите в документации.

# Работа с SQLite

SQLite — это однофайловая реляционная база данных в комплекте с большинством стандартных установок Python. SQLite часто является предпочтительной технологией для небольших приложений, особенно для встроенных систем и устройств, таких как телефоны и планшеты, интеллектуальные приборы и инструменты.

## Установка и подключение к БД

Создать новую базу данных SQLite так же просто, как создать соединение с помощью модуля sqlite3 в стандартной библиотеке Python. Чтобы установить соединение, нужно передать путь к файлу методу *connect(...)* в модуле sqlite3, и, если база данных, представленная файлом, не существует, она будет создана по этому пути.

| import sqlite3  con = sqlite3.connect(‘sqlite.db’) |
| --- |

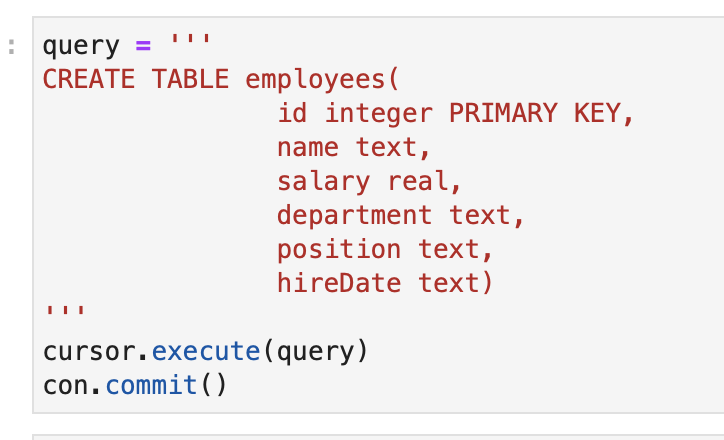
Для выполнения операторов SQL нужен объект курсора, создаваемый методом *cursor().*

Курсор SQLite3 — это метод объекта соединения. Для выполнения операторов SQLite3 сначала устанавливается соединение, а затем создаётся объект курсора с использованием объекта соединения следующим образом:

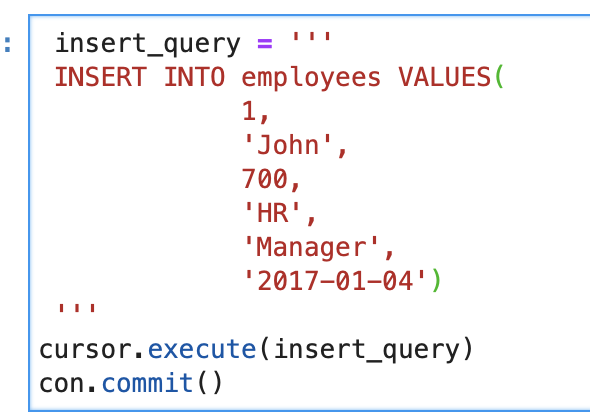


Чтобы создать таблицу в SQLite3, выполним запрос *CREATE TABLE* в методе execute(). Для этого, используя объект курсора, вызывается метод *execute* с запросом *create table* в качестве параметра.

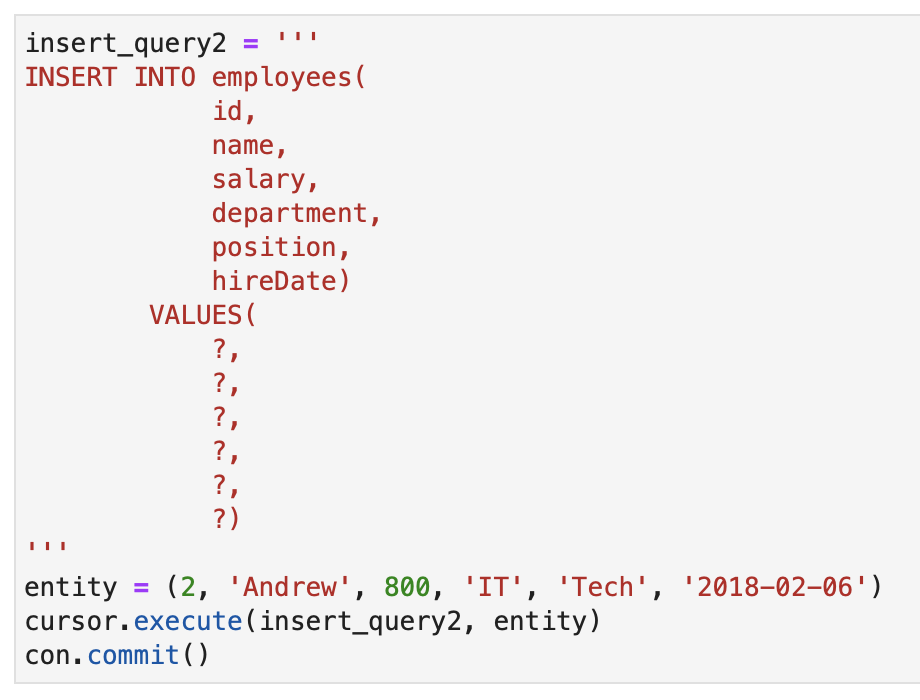
Давайте создадим таблицу Employees со следующими колонками: employees (id, name, salary, department, position, hireDate). При этом для каждой колонки мы укажем тип данных. Для колонки id мы ещё указали *PRIMARY KEY* — это первичный ключ, ограничение, позволяющее однозначно идентифицировать каждую запись в таблице SQL.



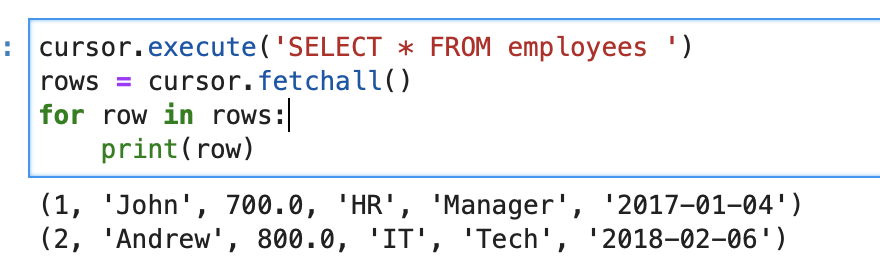
Метод *commit()* сохраняет все сделанные изменения. Чтобы вставить данные в таблицу, воспользуемся оператором *INSERT INTO*.



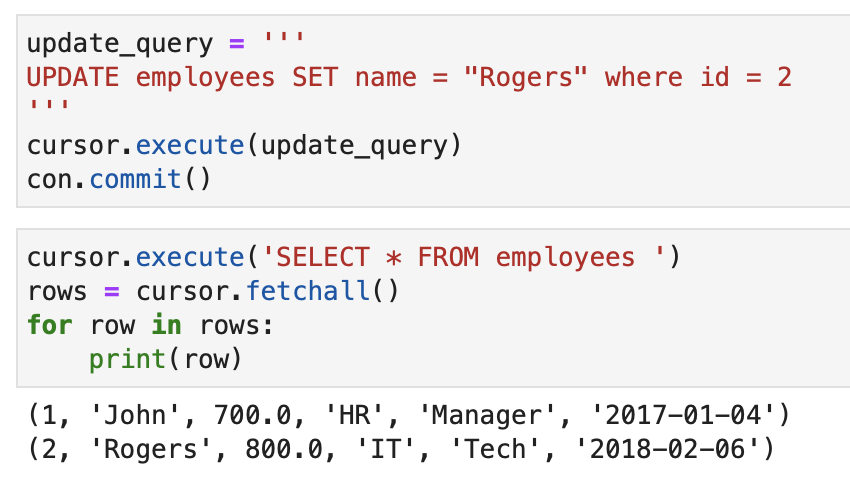
Также можем передать значения / аргументы в оператор *INSERT* в методе *execute()*. Также можно использовать знак вопроса (?) в качестве заполнителя для каждого значения. Синтаксис *INSERT* будет выглядеть следующим образом:



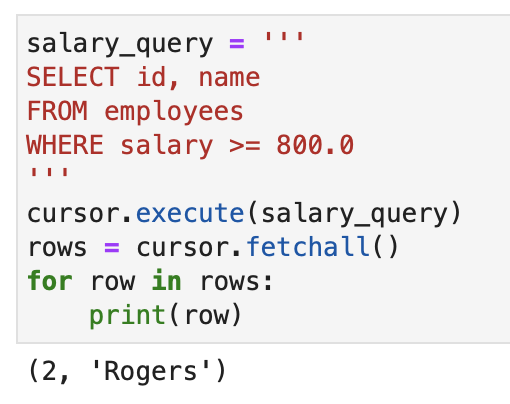
Чтобы извлечь данные из БД, выполним инструкцию *SELECT*, а затем воспользуемся методом *fetchall()* объекта курсора для сохранения значений в переменной. При этом переменная будет являться списком, где каждая строка из БД будет отдельным элементом списка. Далее будет выполняться перебор значений переменной и печатать значений. Посмотрим, какая таблица у нас получилась.



Предположим, что нужно обновить имя сотрудника, чей идентификатор равен 2. Для обновления будем использовать инструкцию *UPDATE*. Также воспользуемся предикатом *WHERE* в качестве условия для выбора нужного сотрудника.



Теперь выберем имена и идентификаторы тех сотрудников, у кого зарплата больше или равна 800:



Счётчик строк SQLite3 используется для возврата количества строк, которые были затронуты или выбраны последним выполненным запросом SQL.

Когда вызывается *rowcount* с оператором *SELECT*, будет возвращено -1, поскольку количество выбранных строк неизвестно до тех пор, пока все они не будут выбраны. Рассмотрим пример:



Поэтому, чтобы получить количество строк, нужно получить все данные, а затем получить длину результата:



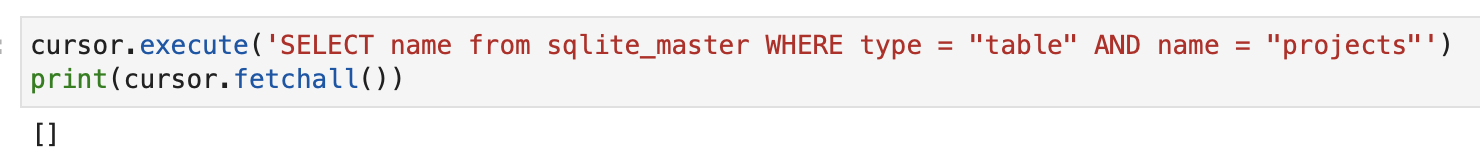
При создании таблицы нужно убедиться, что таблица ещё не существует. Аналогично при удалении таблицы — она должна существовать.

Чтобы проверить, если таблица ещё не существует, используем *if not exists* с оператором *CREATE TABLE* следующим образом:



Теперь давайте посмотрим, какие таблицы у нас есть в базе данных:



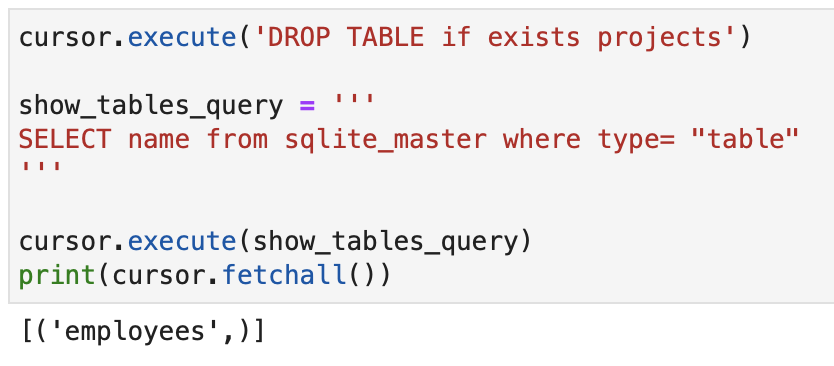
Также проверим, существует ли таблица, к которой нужно получить доступ, выполнив запрос:

Если указанное имя таблицы не существует, вернётся пустой массив.

Удаление таблицы выполняется с помощью оператора *DROP*. Его cинтаксис:

| drop table table\_name |
| --- |

Так же, чтобы проверить, существует ли таблица при удалении, мы используем *if exists* с инструкцией *DROP TABLE* следующим образом:



Для вставки нескольких строк одновременно используют оператор **executemany**.

Здесь мы создали таблицу с двумя столбцами, тогда у «данных» есть четыре значения для каждого столбца. Эта переменная передаётся методу *executemany()* вместе с запросом.

Обратите внимание, что использовался заполнитель для передачи значений.

Когда работа с БД завершена, рекомендуется закрыть соединение. Соединение может быть закрыто с помощью метода *close()*. Чтобы закрыть соединение, используйте объект соединения с вызовом метода *close()* следующим образом:



# Домашнее задание

1. Сохранить данные из предыдущего домашнего задания в файл .json или .csv.
2. Создать MongoDB, записать данные туда (любое название базы, любое название коллекции). Выполнить команду для демонстрации содержимого коллекции. Прикрепить скриншот.
3. Создать базу данных sqlite, загрузить туда данные из парсера с предыдущего урока. Загрузить файл .db

# Глоссарий

**СУБД** (DBMS, Database Management System) — система управления базами данных. Это набор команд, прикладных и инфраструктурных приложений и библиотек, которые позволяют управлять и обслуживать базу данных.

**SQL** (Structured Query Language) — декларативный язык программирования, применяемый для создания, модификации и управления данными в реляционной базе данных, управляемой соответствующей системой управления базами данных.

**NoSQL** — ряд подходов, направленных на реализацию систем управления базами данных, имеющих существенные отличия от моделей, используемых в традиционных реляционных СУБД с доступом к данным средствами языка SQL.

**MongoDB** — документоориентированная СУБД с открытым исходным кодом, не требует схемы таблиц и относится к NoSQL. MongoDB хранит данные в виде документов в формате JSON.

**SQLite** — это встраиваемая кроссплатформенная БД, которая поддерживает достаточно полный набор команд SQL и доступна в исходных кодах (на языке C).

# Дополнительные материалы

1. [Руководство по MongoDB](https://proselyte.net/tutorials/mongodb/introduction/).
2. [Руководство по Sqlite3](https://docs.python.org/3/library/sqlite3.html).