

Лекция 5. Модели



В лекции будут рассмотрены вопросы:

Модели данных в приложениях Django

СУБД SQLite и приложение SQLiteStudio

Реляционная модель данных

Настройка параметра DATABASES в файле settings.py

Описание моделей в файле models.py

Создание таблиц. Миграция БД. Каталог migrations

Типы полей таблиц БД

Операции с моделями – CRUD

Получение данных из базы

Работа с объектами модели:

- создание и получение объектов модели
- редактирование и удаление объектов модели

Связи между таблицами. Виды связей:

- «один ко многим»,
- «многие ко многим»,
- «один к одному».

Административная панель Django Admin.

Работа с данными в Django Admin:

- Регистрация моделей в файле admin.py
- Создание суперпользователя



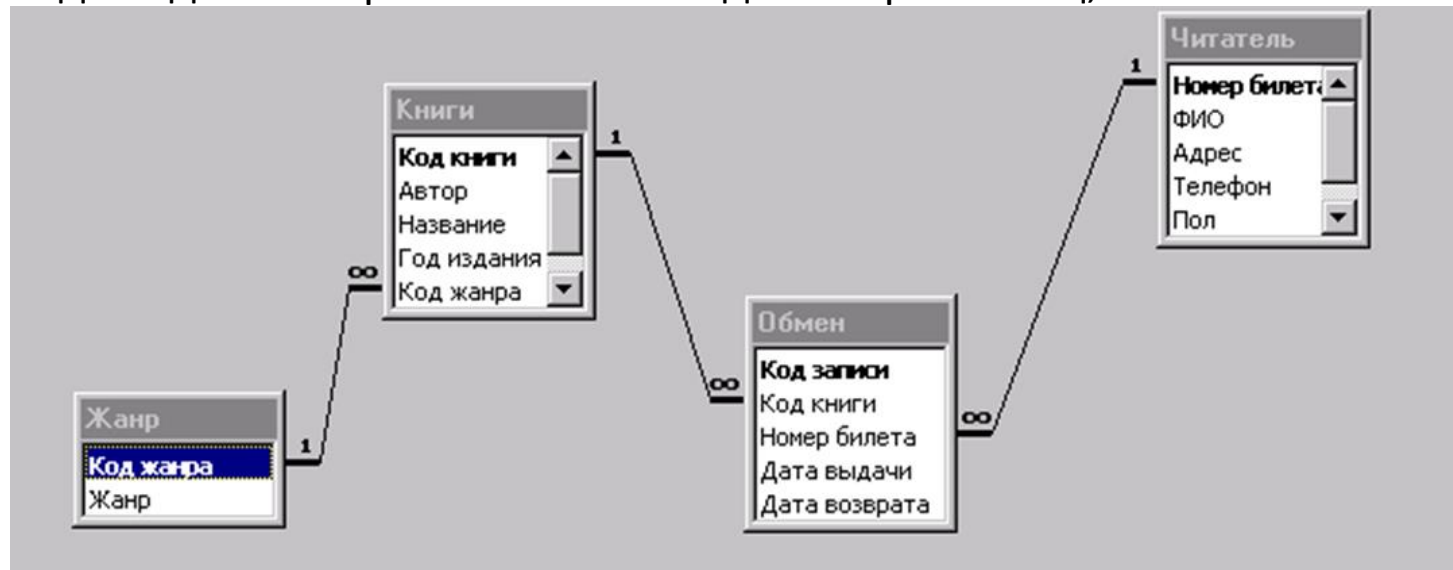
Модели данных в приложениях Django

Специальные программные среды для работы с базами данных называются Системы управления базами данных (СУБД). СУБД используют и приложения Django.

Способ организации данных в СУБД называется моделью данных. СУБД используют разные модели данных: реляционные, сетевые, иерархические.

Наиболее распространенной моделью данных, поддерживаемой многими СУБД, в том числе и SQLite, встроенной в Django, является реляционная (от слова relation -отношение или таблица) модель данных.

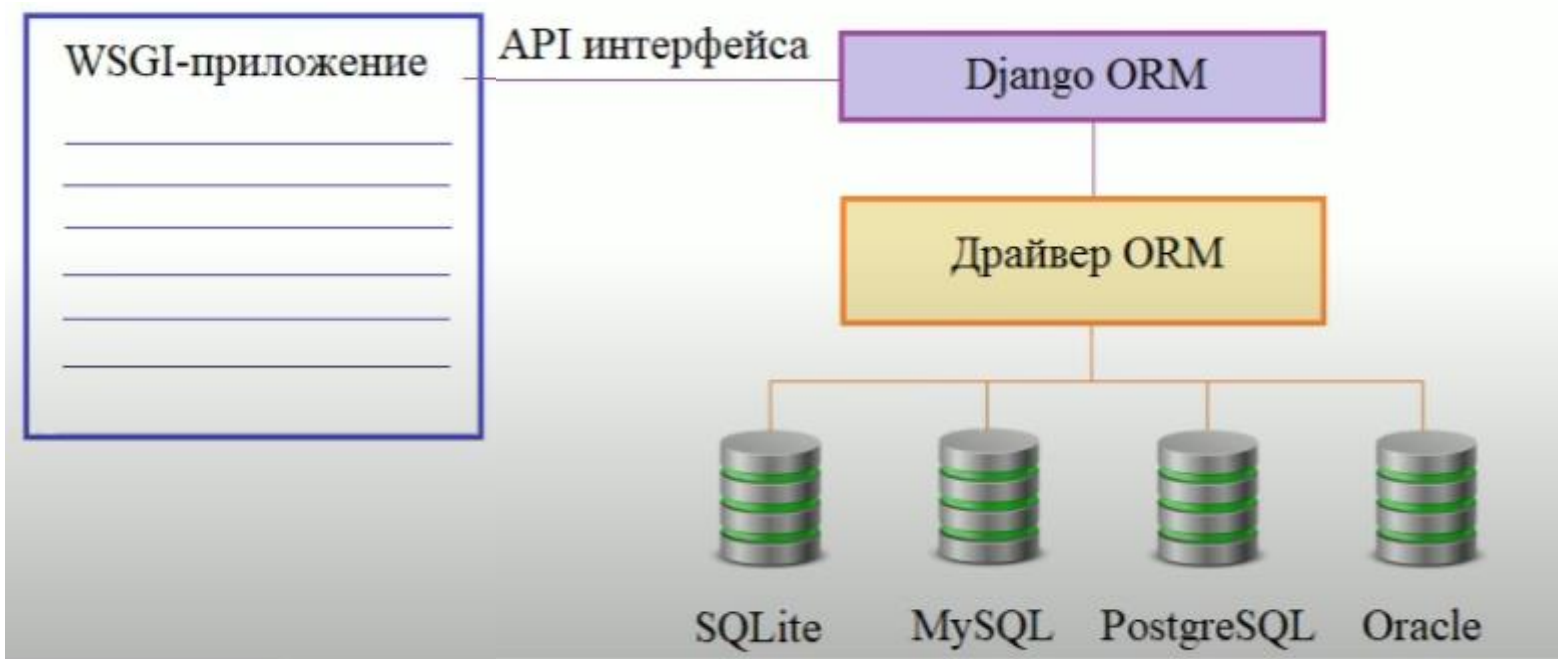
В реляционной модели данные организованы в виде набора таблиц, связанных по ключевым атрибутам.



Модели – это часть паттерна MVC, поддерживаемого Django.

Приложения Django могут использовать различные СУБД: SQLite, MySQL, PostgreSQL, Oracle и другие. В процессе разработки приложения СУБД можно менять. Можно ли построить универсальную программу, не привязанную к конкретной СУБД?

Для этого в Django встроен механизм взаимодействия с таблицами баз данных через объекты классов языка Python посредством технологии ORM (Object-Relational Mapping) - объектно-реляционное отображение.



Приложение через API-интерфейс ORM Django может взаимодействовать с любым типом базы данных. При этом программный код на уровне WSGI-приложения будет оставаться универсальным, независимым от выбранной СУБД.



По умолчанию Django сконфигурирован для работы с СУБД SQLite. Для работы с другими СУБД необходимы соответствующие драйверы.

В файле **settings.py** в проекте Django для работы с базами данных определен словарь DATABASES, который по умолчанию выглядит так:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Параметр ENGINE указывает на используемый движок для доступа к базе данных.

В данном случае это встроенный пакет django.db.backends.sqlite3.

Параметр NAME указывает на путь к базе данных. После первого запуска проекта в нем по умолчанию будет создан файл db.sqlite3, который и будет представлять базу данных.



Описание моделей в файле `models.py`

При создании приложения Django в его каталог по умолчанию добавляется файл `models.py`, в котором и описывается модель данных. В модели каждую таблицу представляет класс, унаследованный от объекта `django.db.models.Model`. Например:

```
class Language(models.Model):
```

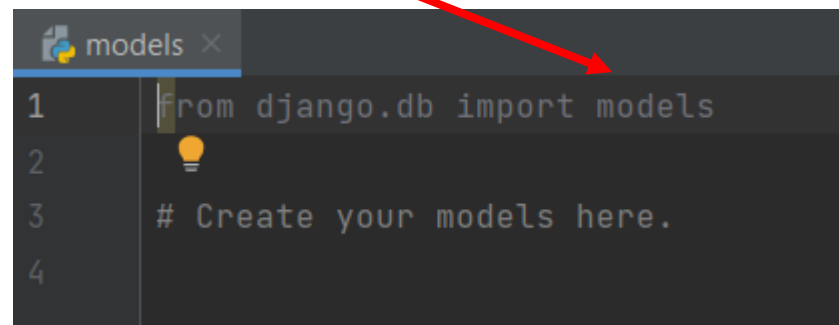
```
    name = models.CharField(max_length=20, help_text=" Введите язык книги", verbose_name=" Язык книги")
```

```
    def __str__(self):
```

```
        return self.name
```

Первоначально файл `models.py` содержит только импорт пакета `models` и приглашение разработчику описать свою модель данных:

Пакет `models` содержит базовые классы моделей, на основе которых можно создавать свои модели таблиц базы данных.



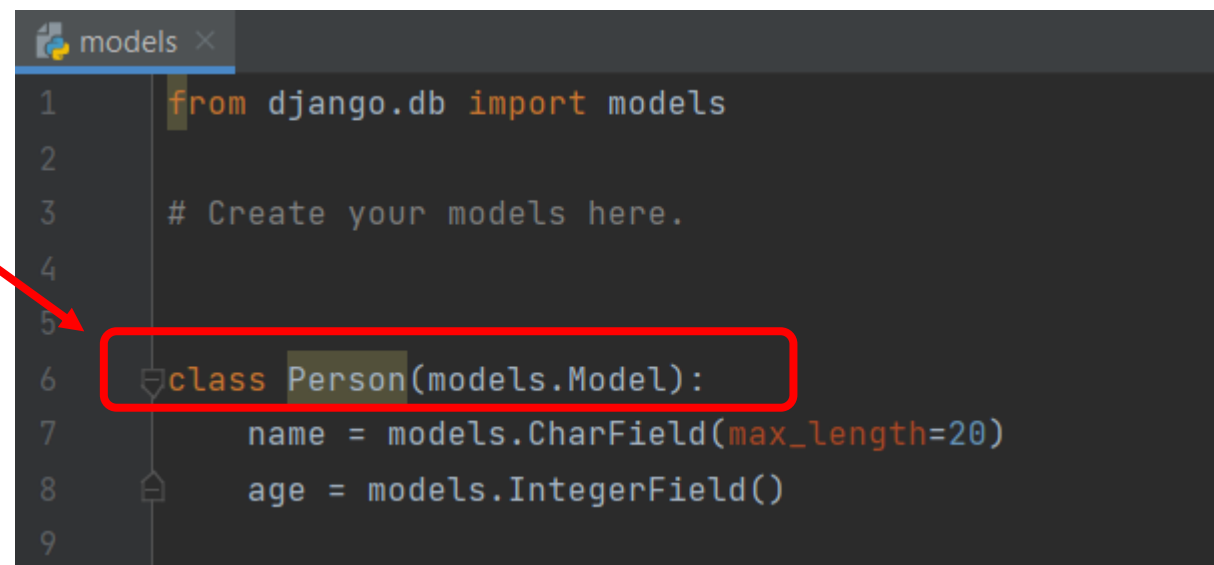
В файле **models.py** опишем модель, которая пока будет содержать одну таблицу. Таблица будет представлена классом **Person**, наследующим от базового класса Model, который содержит все механизмы, необходимые для того, чтобы разработчик мог создавать свои классы моделей.

Определим в таблице два поля:

Поле name (тип CharField)- текстовое поле, для хранения имени. Для типа CharField обязательно надо указывать параметр max_length, который задает максимальную длину хранящейся строки.

Поле age (тип IntegerField) - числовое поле для хранения возраста - целые числа.

Поле **Id** при этом не описываем, оно уже прописано в классе Model.



```
models x
1  from django.db import models
2
3  # Create your models here.
4
5
6  class Person(models.Model):
7      name = models.CharField(max_length=20)
8      age = models.IntegerField()
9
```

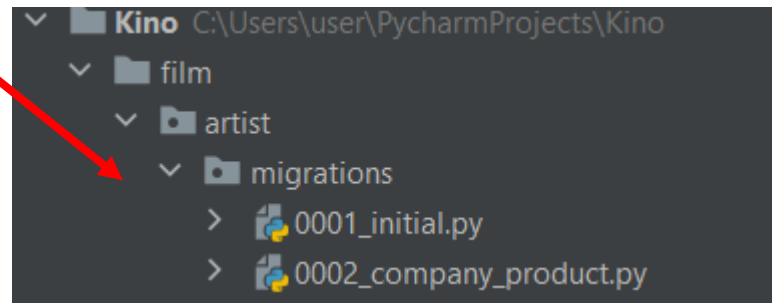


Миграции

Операция преобразования базы данных в соответствии с описанием модели данных называется **миграцией**. При выполнении миграции в базе данных создаются новые или изменяются прежние таблицы.

Миграции – это модули языка Python, где описаны наборы команд на уровне ORM-интерфейса для создания таблиц определенных структур.

При каждом изменении модели данных создается очередная миграция и сохраняется в виде файла с номером в папке **migrations** приложения.



Миграция выполняется в два этапа.

1 этап. Создание миграции.

В окне терминала ввести команду:

```
python manage.py makemigrations
```

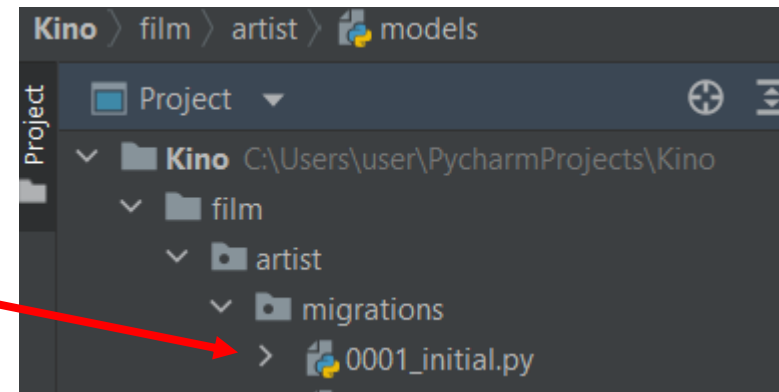
После выполнения команды будет создан файл миграции `artist/migrations/0001_initial.py` и выведено сообщение:

```
Create model Person.
```



В папке migrations создан файл миграции
0001_initial.py

и в окне терминала выведено сообщение
Create model Person:



```
PS C:\Users\user\PycharmProjects\Kino> cd film
PS C:\Users\user\PycharmProjects\Kino\film> python manage.py makemigrations
Migrations for 'artist':
  artist\migrations\0001_initial.py
    - Create model Person
PS C:\Users\user\PycharmProjects\Kino\film>
```



Файл 0001_initial.py будет содержать:

```
5
6 class Migration(migrations.Migration):
7
8     initial = True
9
10    dependencies = [
11    ]
12
13    operations = [
14        migrations.CreateModel(
15            name='Person',
16            fields=[
17                ('id', models.BigAutoField(auto_created=True, primary_key=True, serialize=False, verbose_name='ID')),
18                ('name', models.CharField(max_length=20)),
19                ('age', models.IntegerField()),
20            ],
21        ),
22    ]
```

Важно! В структуре таблицы добавилось поле id – для значения уникального идентификатора каждой записи.

Такие ключевые поля не требуют предварительного описания и создаются автоматически.



Можно посмотреть SQL-запрос, который будет выполнен для создания таблицы. Для этого в режиме терминала надо ввести: **python manage.py sqlmigrate artist 0001**

```
PS C:\Users\user\PycharmProjects\Kino> cd film
PS C:\Users\user\PycharmProjects\Kino\film> python manage.py sqlmigrate artist 0001
BEGIN;
--
-- Create model Person
--
CREATE TABLE "artist_person" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "name" varchar(20) NOT NULL, "age" integer NOT NULL);
COMMIT;
PS C:\Users\user\PycharmProjects\Kino\film>
```



Важно! В структуре таблицы добавилось поле `id` – для значения уникального идентификатора каждой записи. Такие ключевые поля не требуют описания и создаются автоматически.

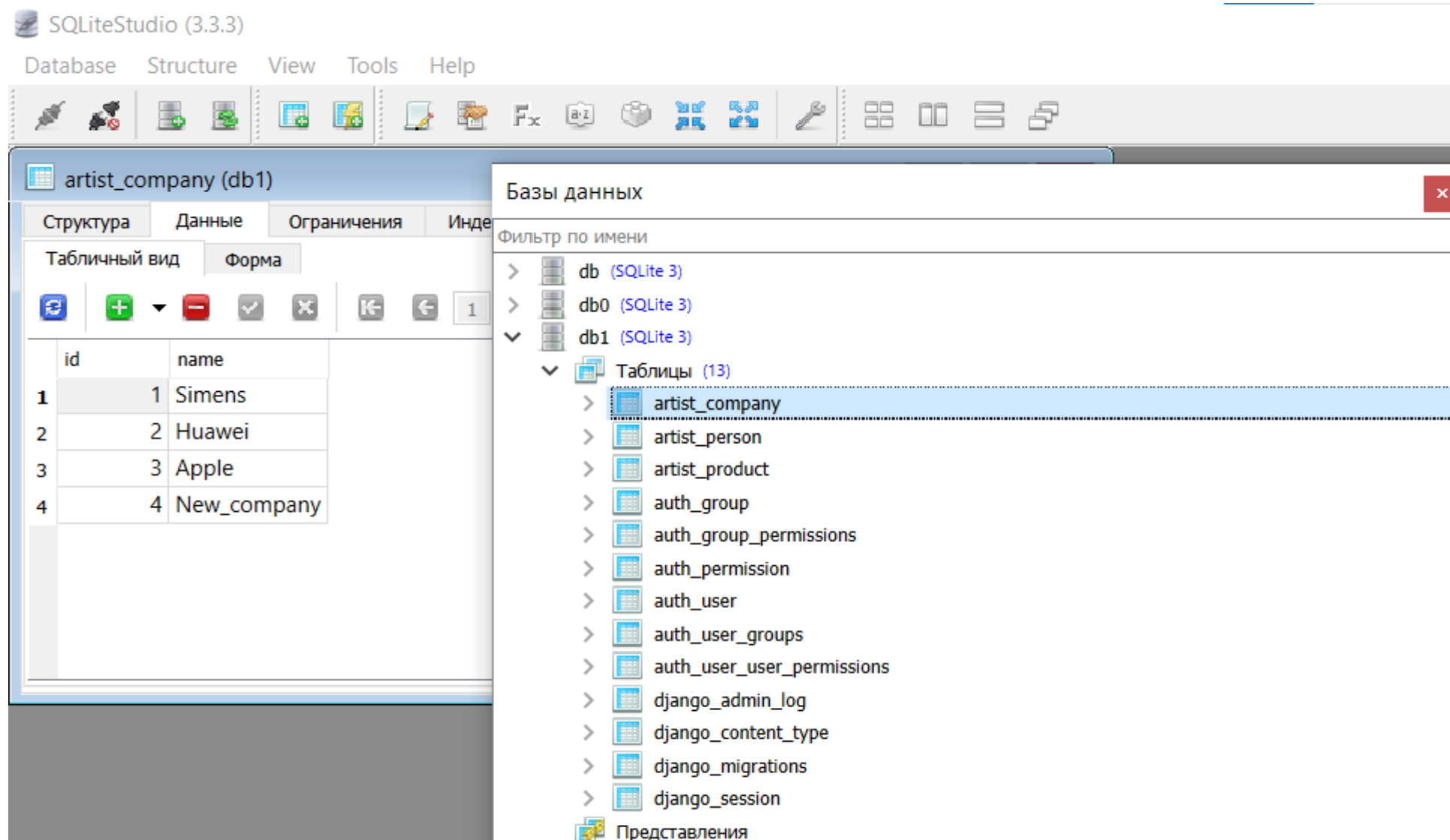
2 этап. Выполнение миграции. В окне терминала введем:

```
python manage.py migrate
```

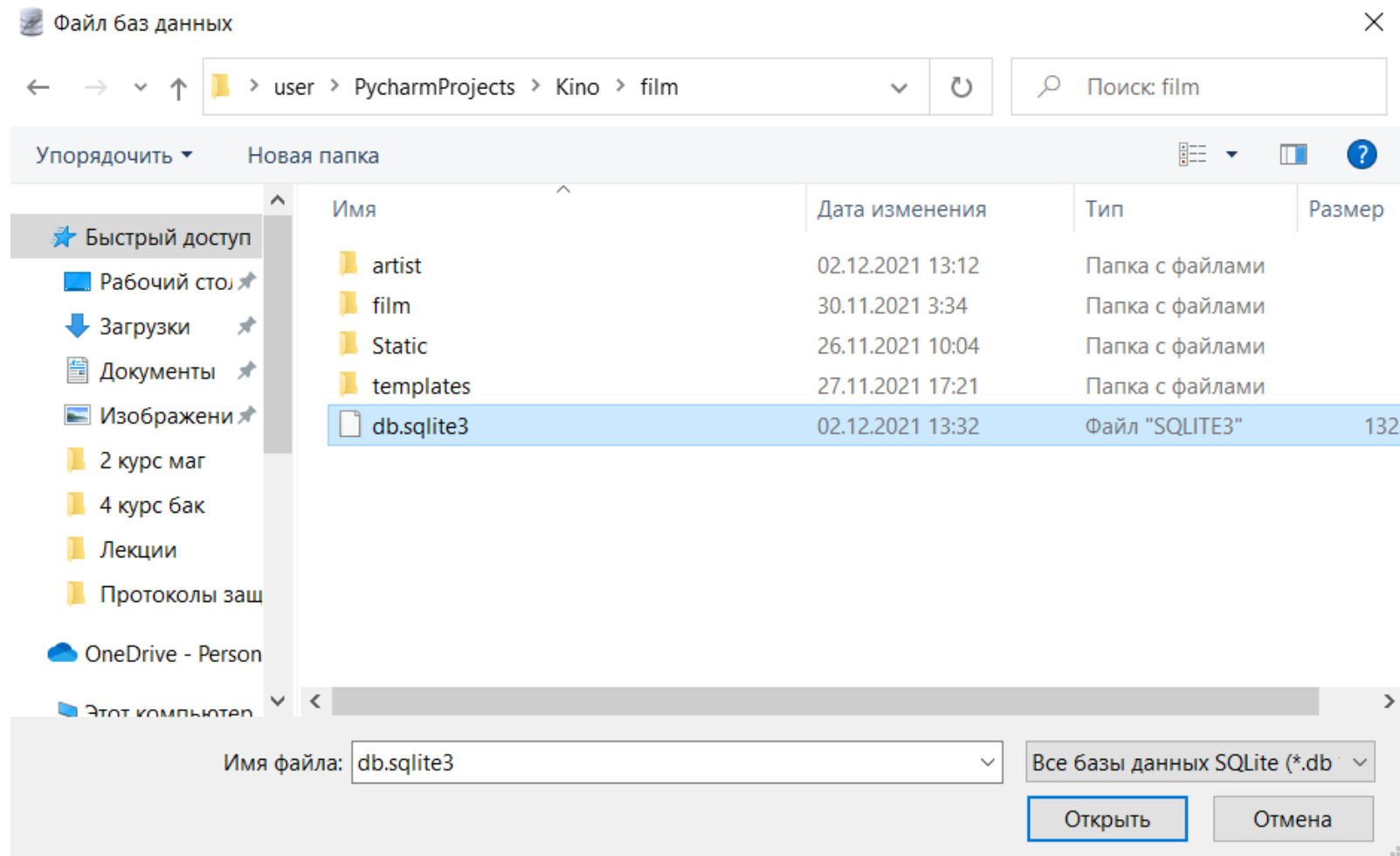
В результате этого таблица в базе будет создана.



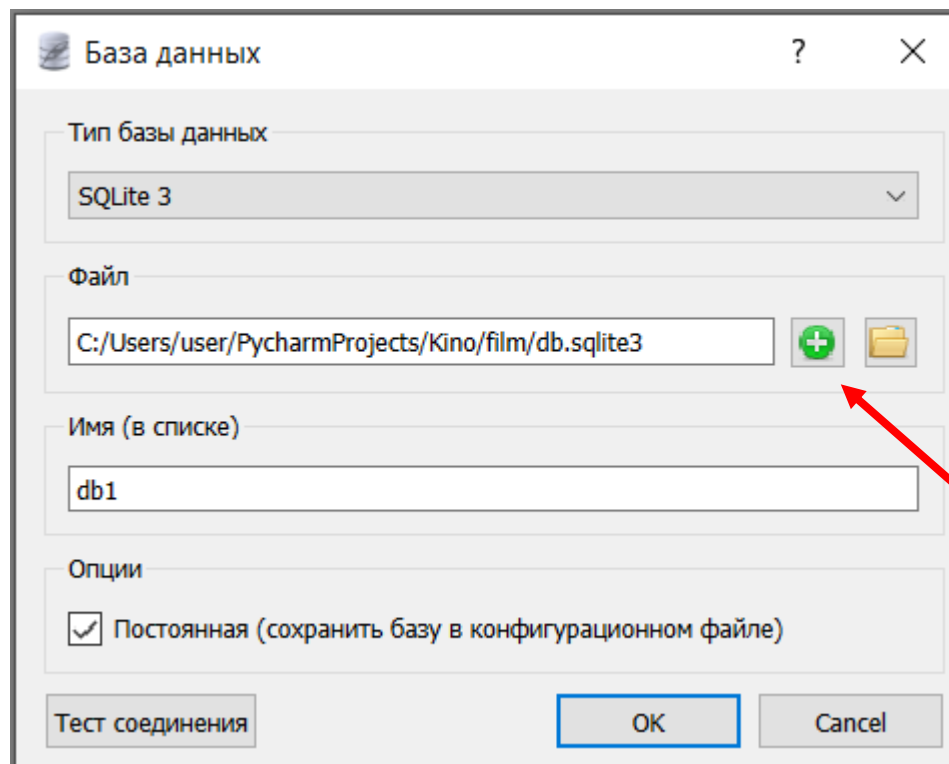
Для демонстрации данных таблиц приложения удобно использовать программу SQLiteStudio:



Откроем базу db.sqlite3 в среде SQLiteStudio. Выберем в меню: Database/Add a database, папку в строке Файл и выберем базу нашего проекта.



Подключаем базу:



The screenshot displays the SQLiteStudio 3.3.3 interface. The main window shows the 'artist_person' table structure for database 'db1'. The 'Структура' (Structure) tab is active, showing a table with three columns: 'id' (integer, primary key), 'name' (varchar(20)), and 'age' (integer). The 'Данные' (Data) tab is also visible. A red arrow points to the 'Данные' tab, and another red arrow points to the 'artist_person' table in the database list on the right.

artist_person (db1)

Структура Данные Ограничения Индексы Триггеры DDL

db1 Имя таблицы: artist_person ☐ WITHOUT ROWID

	Имя	Тип данных	Первичный ключ	Внешний ключ	Уникальность	Пр...
1	id	integer				
2	name	varchar (20)				
3	age	integer				

Тип Имя Подробности

Базы данных

Фильтр по имени

- > db (SQLite 3)
- > db0 (SQLite 3)
- ▼ db1 (SQLite 3)
 - ▼ Таблицы (11)
 - > artist_person
 - > auth_group
 - > auth_group_permissions
 - > auth_permission
 - > auth_user
 - > auth_user_groups
 - > auth_user_user_permissions
 - > django_admin_log
 - > django_content_type
 - > django_migrations
 - > django_session
 - Представления



Вкладка Структура таблицы artist_person:





SQLiteStudio (3.3.3)

Database Structure View Tools Help

artist_person (db1)

Структура Данные Ограничения Индексы Триггеры DDL

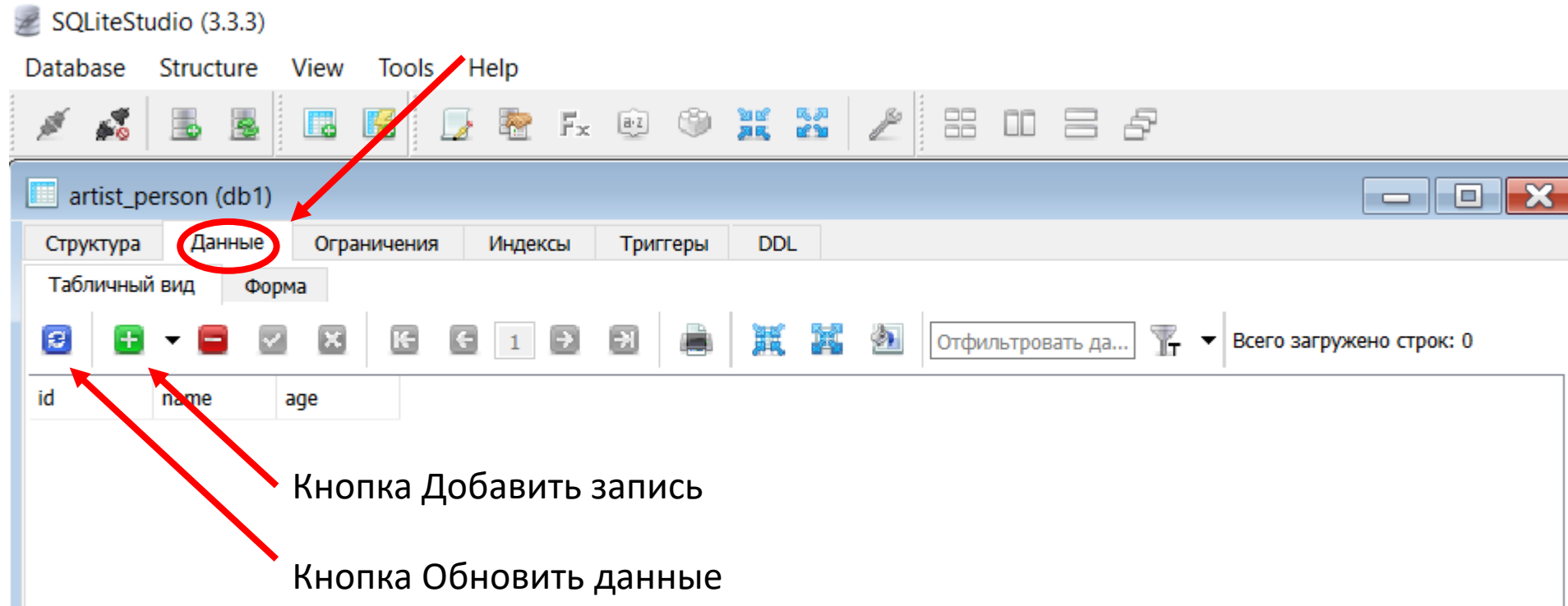
db1 Имя таблицы: artist_person ☐ WITHOUT ROWID

	Имя	Тип данных	Первичный ключ	Внешний ключ	Уникальность	Проверка	Не NULL	Сравнение	Generated	Значение по умолчанию
1	id	integer								NULL
2	name	varchar (20)								NULL
3	age	integer								NULL

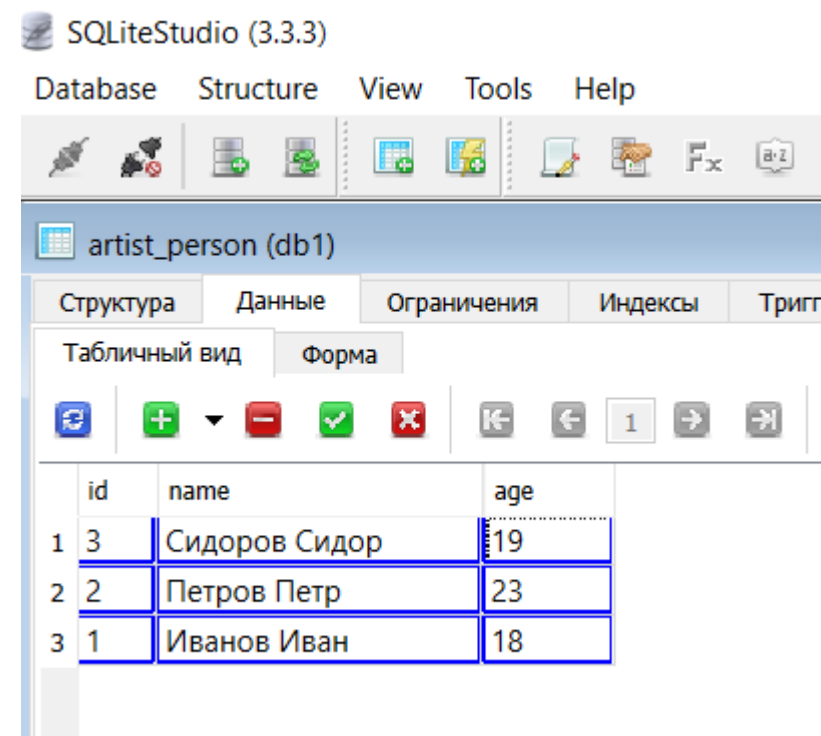
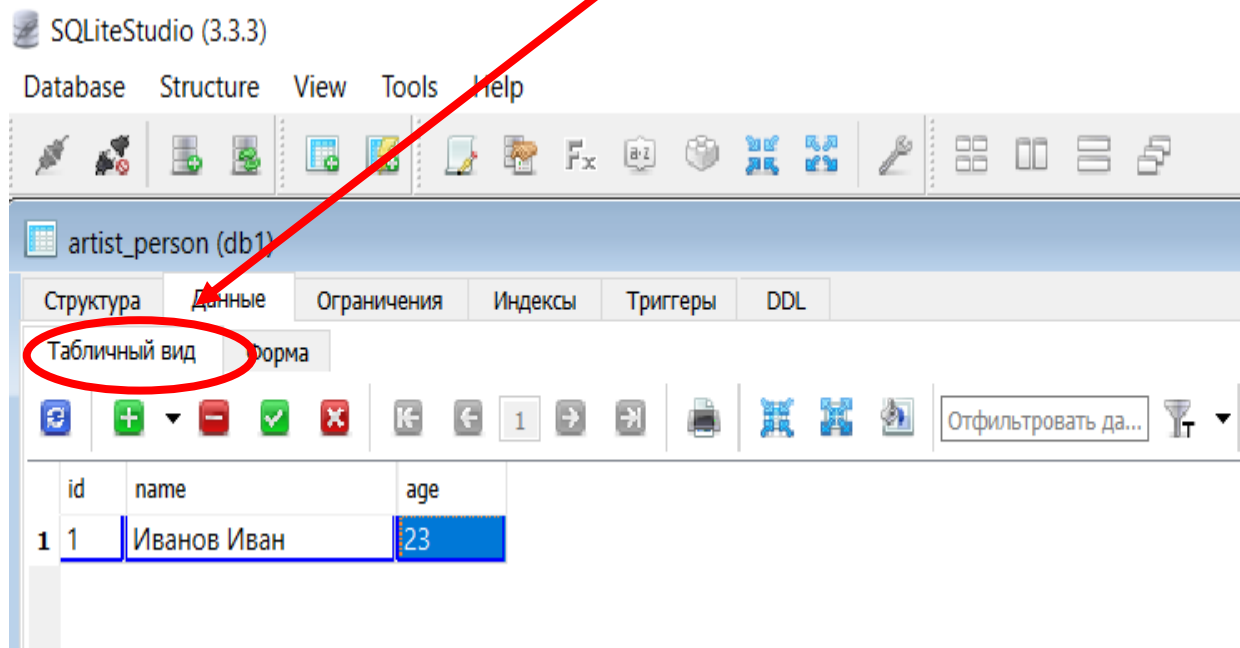
Тип Имя Подробности



Вкладка Данные таблицы artist_person:



Добавляем запись в табличном виде:



Добавляем запись в режиме Формы:

SQLiteStudio (3.3.3)

Database Structure View Tools Help

artist_person (db1)

Структура Данные Ограничения Индексы Триггеры DDL

Табличный вид **Форма**

Всего загружено строк: 0 Строка: 1

id (integer) ☐ Значение Null

Число Текст Шестнадцатеричное

1

name (varchar) ☐ Значение Null

Текст Шестнадцатеричное

Иванов Иван

age (integer) ☐ Значение Null

Число Текст Шестнадцатеричное

23



Типы полей СУБД SQLite

При описании структуры таблиц можно использовать типы полей:

BinaryField(): хранит бинарные данные

BooleanField(): хранит значение True или False (0 или 1)

NullBooleanField(): хранит значение True или False или Null

DateField(): хранит дату

TimeField(): хранит время

DateTimeField(): хранит дату и время

DurationField(): хранит период времени



AutoField(): хранит целочисленное значение, которое автоматически инкрементируется, обычно применяется для первичных ключей

BigIntegerField(): представляет число - значение типа Number, которое укладывается в диапазон от -9223372036854775808 до 9223372036854775807. В зависимости от выбранной СУБД диапазон может немного отличаться

DecimalField(decimal_places=X, max_digits=Y): представляет значение типа Number, которое имеет максимум X разрядов и Y знаков после запятой

FloatField(): хранит, значение типа Number, которое представляет число с плавающей точкой

IntegerField(): хранит значение типа Number, которое представляет целочисленное значение



PositiveIntegerField(): хранит значение типа Number, которое представляет положительное целочисленное значение (от 0 до 2147483647)

PositiveSmallIntegerField(): хранит значение типа Number, которое представляет небольшое положительное целочисленное значение (от 0 до 32767)

SmallIntegerField(): хранит значение типа Number, которое представляет небольшое целочисленное значение (от -32768 до 32767)

CharField(max_length=N): хранит строку длиной не более N символов

TextField(): хранит строку неопределенной длины

EmailField(): хранит строку, которая представляет email-адрес. Значение автоматически валидируется встроенным валидатором EmailValidator



FileField(): хранит строку, которая представляет имя файла

FilePathField(): хранит строку, которая представляет путь к файлу длиной в 100 символов

ImageField(): хранит строку, которая представляет данные об изображении

GenericIPAddressField(): хранит строку, которая представляет IP-адрес в формате IP4v или IP6v

SlugField(): хранит строку, которая может содержать только буквы в нижнем регистре, цифры, дефис и знак подчеркивания

URLField(): хранит строку, которая представляет валидный URL-адрес

UUIDField(): хранит строку, которая представляет UUID-идентификатор



Операции с моделями (CRUD). Методы работы с данными.

Метод `create()`:

Применяется для добавления данных в базу:

```
ivan = Person.objects.create(name="Ivan", age=25)
```

Метод `save()`

Сохраняет данные в базе:

```
ivan=Person(name="Ivan", age=23)
```

```
ivan.save()
```



Получение данных из базы. Получение одного объекта

Метод `get()` возвращает один объект по определенному условию, которое передается в качестве параметра:

```
ivan = Person.objects.get(name="Ivan") # получаем запись, где name="Ivan"
```

```
Jim = Person.objects.get(age=34) # получаем запись, где age=34
```

```
Dirk = Person.objects.get(name="Dirk", age=28) # запись, где name="Dirk" и age=28
```

Если в таблице не окажется подобного объекта, то мы получим ошибку `имя_модели.DoesNotExist`.

Если же в таблице будет несколько объектов, которые соответствуют условию, то будет сгенерировано исключение `MultipleObjectsReturned`.



Метод `get_or_create()` возвращает объект, если он есть, а если его нет в базе, то добавляет в базу новый объект.

```
bob, created = Person.objects.get_or_create(name="Bob", age=24)
print(bob.name)
print(bob.age)
```

Метод возвращает добавленный объект (в данном случае переменная `bob`) и булево значение (`created`), которое хранит `True`, если добавление прошло успешно.



Метод all()

Применяется для получения всех объектов модели:

```
people = Person.objects.all()
```

Метод filter()

Применяется, если надо получить все объекты, которые соответствуют определенному критерию. Критерий выборки принимается в качестве параметра:

```
people = Person.objects.filter(age=23)
```

можно использовать нескольких критериев:

```
people2 = Person.objects.filter(name="Tom", age=23)
```



Метод `exclude()` – применяется для исключения из выборки записей по критерию, переданному в качестве параметра :

исключаем пользователей, у которых `age=23`

```
people = Person.objects.exclude(age=23)
```

Методы можно комбинировать:

выбираем всех пользователей, у которых `name="Tom"`, кроме тех, у которых `age=23`

```
people = Person.objects.filter(name="Tom").exclude(age=23)
```



Метод `in_bulk()`

применяется для чтения большого количества записей.

В то время как методы `all()`, `filter()` и `exclude()` возвращают объект **QuerySet**, метод `in_bulk` возвращает словарь, где ключи представляют **id** объектов, а значения по этим ключам - собственно эти объекты, то есть в данном случае объекты модели `Person`.

```
# получаем все объекты
people = Person.objects.in_bulk()
for id in people:
    print(people[id].name)
    print(people[id].age)
```

```
# получаем объекты с id=1 и id=3
people2 = Person.objects.in_bulk([1,3])
for id in people2:
    print(people2[id].name)
    print(people2[id].age)
```



Обновление. Метод `save()`

Применяется для сохранения и обновления объекта:

```
bob = Person.objects.get(id=2)
```

```
bob.name = "Bob"
```

```
bob.save()
```

При этом объект обновляется полностью, все его свойства, даже если мы их не изменяли. Если надо обновить только определенные поля, следует использовать параметр **`update_fields`** :

```
bob = Person.objects.get(id=2)
```

```
bob.name = "Bobic"
```

```
bob.save(update_fields=["name"]).
```



Метод `update()`

Другой способ обновления объектов представляет метод **`update()`** в сочетании с методом **`filter()`**, которые вместе выполняются в одном запросе к базе данных:

```
Person.objects.filter(id=2).update(name="Mike")
```



Функция F()

Можно использовать, если надо изменить значение столбца в таблице на основании уже имеющегося значения:

```
from django.db.models import F
Person.objects.all(id=2).update(age = F("age") + 1)
```

В данном случае полю **age** присваивается уже имеющееся значение, увеличенное на единицу.

Метод **update()** обновит все записи в таблице, которые соответствуют условию.

Если надо обновить все записи независимо от условия, можно комбинировать метод **update()** с методом **all()**:

```
from django.db.models import F
Person.objects.all().update(name="Mike")
Person.objects.all().update(age = F("age") + 1)
```



Метод `update_or_create()`

Метод обновляет запись, а если записи еще нет, то добавляет ее в таблицу:

Метод **`update_or_create()`** принимает два параметра.

Первый параметр представляет критерий выборки объектов, которые будут обновляться.

Второй параметр представляет объект со значениями, которые будут переданы тем записям, которые соответствуют критерию из первого параметра.

Если критерию не соответствует никаких записей, то в таблицу добавляется новый объект.

```
values_for_update={"name":"Bob", "age": 31}
```

```
bob, created = Person.objects.update_or_create(id=2, defaults = values_for_update)
```



Удаление. Метод delete()

Удаляет запись из таблицы.

```
person = Person.objects.get(id=2)  
person.delete()
```

Если не требуется получение отдельного объекта из базы данных, тогда можно удалить объект с помощью комбинации **методов filter() и delete()**:

```
Person.objects.filter(id=4).delete()
```



Просмотр строки запроса. Свойство query.

С помощью свойства **query** мы можем получить SQL-запрос, который выполнялся. Например:

```
people = Person.objects.filter(name="Tom").exclude(age=34)  
print(people.query)
```

Данный код отобразит на консоли SQL-запрос типа следующего:

```
SELECT "artist_person"."id", "artist_person"."name", "artist_person"."age"  
FROM "artist_person" WHERE ("artist_person"."name" = Tom AND NOT ("artist_person"."age" =  
34))
```



Конец Части 1. Продолжение следует...

