**СПРИНТ 4.**

**Django: дорабатываем приложение.**

1. **CRUD и фильтрация через ORM**

Для работы с БД у моделей в Django есть встроенный набор методов. Они наследуются от класса models.Model и поддерживают основные операции по обработке данных в БД: **CRUD**. Вы знакомы с этой аббревиатурой из урока по SQL

**CRUD-операции**

* **Create**: Model.objects.create() — создание объекта в базе
* **Read**: Model.objects.get(id=N) — чтение объекта по его ключу
* **Update**: [object.property](http://object.property) = 'new value' и потом object.save() — изменение объекта
* **Delete**: object.delete() — удаление объекта из базы

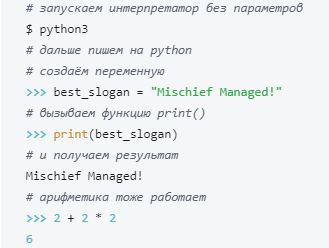
Сейчас мы разберёмся с основными задачами, которые решает Django ORM. Но перед этим познакомимся с инструментом, который упростит нам тестирование кода.

**Python shell**

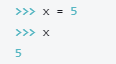
С интерпретатором кода Python, как и со многими другими программами, можно работать через командную строку. Если в консоли выполнить команду $python3 без параметров, то интерпретатор Python запустится в «интерактивном режиме». Теперь можно ввести в командную строку любые скрипты python — и они будут выполняться прямо в терминале. Это похоже на работу командной строки, но вместо команд для работы с файлами выполняется программный код, строчка за строчкой.

Откройте новое окно терминала и посмотрите, как работает python shell.

Символы  >>> — это приглашение для ввода команд, то же, что и знак $ в командной строке.



Прелесть этого режима в том, что он тут же выводит результат выполнения скрипта. Например, создадим и выведем переменную:



Обратите внимание: переменные, которые вы создали во время работы в python shell, будут доступны до тех пор, пока вы не закроете окно терминала.

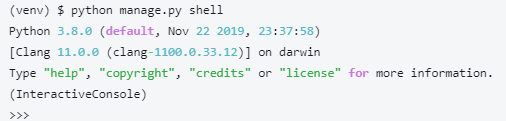
**Django Python shell**

В таком же интерактивном режиме можно работать и с Django-проектами. Для этого python shell надо запустить в виртуальном окружении проекта.

Откройте терминал, убедитесь, что запущено виртуальное окружение проекта Yatube и выполните команду

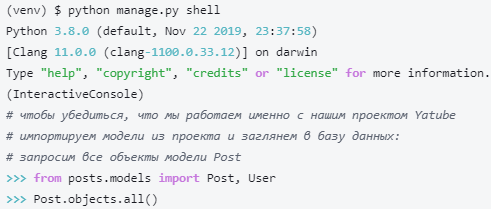
(venv) $ python manage.py shell

Вы увидите примерно такой результат:



Всё, что вы хотите узнать, но стесняетесь спросить — выясняйте через команду help

При работе в интерактивном режиме в Django вам становятся доступны все данные проекта. Можно создавать объекты, управлять базой данных, тестировать функции проекта

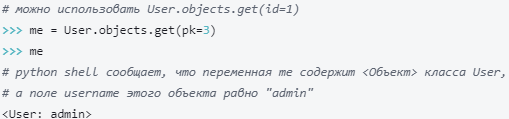


Основы работы с shell разобрали, теперь можно и делами заняться. Все следующие команды выполняйте в shell.

**Работаем с проектом**

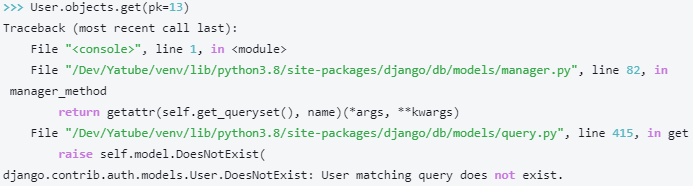
Запрос к базе возвращает специальный объект **QuerySet**, который содержит список объектов, соответствующих условиям запроса. По запросу **.all()** мы получили все объекты модели Post

Чтобы получить определённый объект, можно обратиться к нему по его *primary key*:

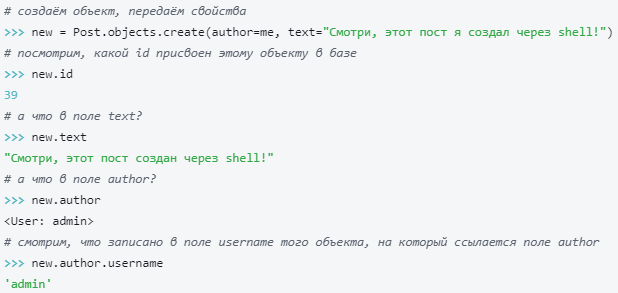


Класс User предустановлен в Django, и для него настроен вывод на экран именно в таком виде. При создании любого класса можно описать, каким образом объекты этого класса будут выводиться на экран (например, в python shell или при вызове print(any\_object)). Это описывается в «магическом методе» \_\_str\_\_

Запросим пользователя с pk=13: у нас в базе такой записи пока что нет. Если объект с запрошенным ключом не найден, то появится сообщение об ошибке:

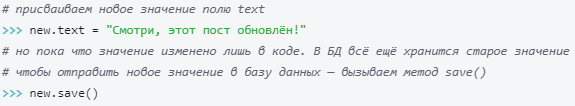


Новый объект в базе можно создать методом **.create():**



Объект new в момент создания сохранился в базе и получил уникальный *id*.

Чтобы изменить этот объект, надо присвоить новое значение одному из его полей и вызвать метод  **save()**:



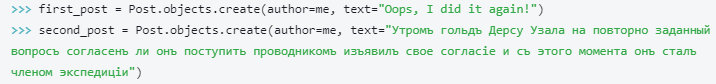
Если вы изменили объект в коде — он не изменится в базе до тех пор, пока вы не вызовете метод  **save().**

Теперь обновлённый текст записи можно увидеть в админ-зоне.

Удалить объект из базы можно методом **delete()**. При вызове этот метод дополнительно удалит и все связанные объекты, для которых был задан параметр on\_delete=models.CASCADE.

**>>>** new.delete()

Для дальнейшей работы нам понадобятся тестовые посты админа, создайте их:



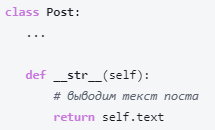
**Фильтрация объектов**

Основная задача при работе с базой — это поиск объектов по заданным признакам. В SQL за это отвечают команды блока WHERE, в Django ORM — метод **filter()**:

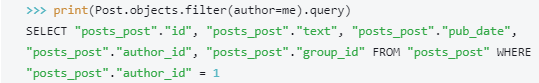


В базе найдены две записи, соответствующие условиям запроса.

Если ваш вывод выглядит так <QuerySet [<Post: Post object(39)>, <Post: Post object(40)>] значит в классе Post не хватает метода \_\_str\_\_ , который нужен для красивого вывода объектов на экран. Добавьте его:



Увидеть SQL-запрос, который будет отправлен к базе, можно с помощью команды **.query**:

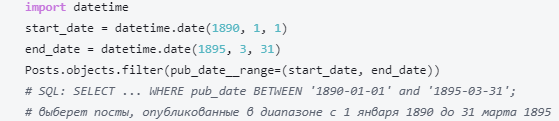


В Django ORM аналог команд WHERE выглядит так: указывается имя поля, затем два знака подчеркивания \_\_, название фильтра и его значение:

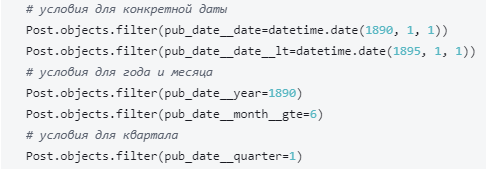


При запросе указываются именованные параметры функции filter(). Имя параметра состоит из имени поля и суффикса, указывающего, какой оператор применять. Доступные операторы:

* **exact** — точное совпадение. «Найти пост, где поле *id* точно равно 1» ORM: Post.objects.filter(id\_\_exact=1) или Post.objects.filter(id=1) На SQL это условие выглядит так: SELECT ... WHERE id = 1. Сравнение работает и с None. Выражение Post.objects.filter(text=None) превратится в SELECT ... WHERE text IS NULL
* **contains** — поиск по тексту в поле text. «Найти пост, где в поле *text* есть слово "oops" именно в таком регистре» ORM: Post.objects.filter(text\_\_contains="oops") SQL: SELECT ... WHERE text LIKE '%oops%'; Ничего не найдётся: регистр не совпадает со словом в посте админа.
* **in** — вхождение в множество. «Найти пост, где значение поля *id* точно равно одному из значений: 1, 3 или 4» ORM: Post.objects.filter(id\_\_in=[1, 3, 4]) SQL: SELECT ... WHERE id IN (1, 3, 4); Если вместо списка будет передана строка, она разобьётся на символы: «Найти пост, где значение поля *text* точно равно "o", "p" или "s"» ORM: Post.objects.filter(id\_\_in="oops") SQL: SELECT ... WHERE text IN ('o', 'p', 's');
* Операторы сравнения **gt** — > (больше), **gte** — => (больше или равно), **lt** — < (меньше), **lte** — <= (меньше или равно). «Найти пост, где значение поля *id* больше пяти» ORM: Post.objects.filter(id\_\_gt=5) SQL: SELECT ... WHERE id>5;
* Операторы сравнения с началом и концом строки **startswith**, **endswith** «Найти посты, где содержимое поля *text* начинается со строки "Утромъ"» ORM: Post.objects.filter(text\_\_startswith="Утромъ") SQL: SELECT ... WHERE text LIKE Утромъ% ESCAPE
* **range** — вхождение в диапазон



* При работе с частями дат можно применять дополнительные суффиксы **date**, **year**, **month**, **day**, **week, week\_day, quarter**, и указывать для них дополнительные условия:

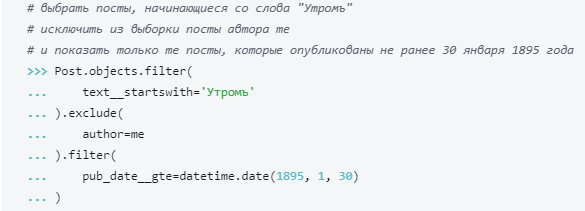


Такой же синтаксис применяется и для времени: **hour**, **minute**, **second**.

* **isnull** — проверка на пустое значение. ORM: Post.objects.filter(pub\_date\_\_isnull=True) SQL: SELECT ... WHERE pub\_date IS NULL;

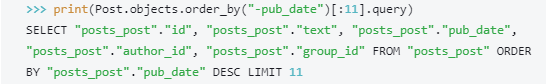
**Объединение условий**

В одном запросе можно указать несколько условий одновременно. Для этого последовательно вызовите методы filter() с различными параметрами. Будет сгенерирован SQL-запрос, в котором все условия объединены оператором AND. Исключить данные из выборки можно методом exclude()



**Сортировка и ограничение количества результатов**

Этот синтаксис вам знаком: мы применяли сортировку во view-функции index(), и там же ограничили число возвращаемых результатов запроса. .order\_by("-pub\_date") — сортировать результаты по полю pub\_date в обратном порядке (от больших значений к меньшим) [:11] — вернуть не более одиннадцати результатов из найденных.



Сортировку и ограничение числа возвращаемых результатов можно объединить с фильтрацией



В Django ORM есть и дополнительный синтаксис, он описан в [документации](https://docs.djangoproject.com/en/3.0/topics/db/queries/).