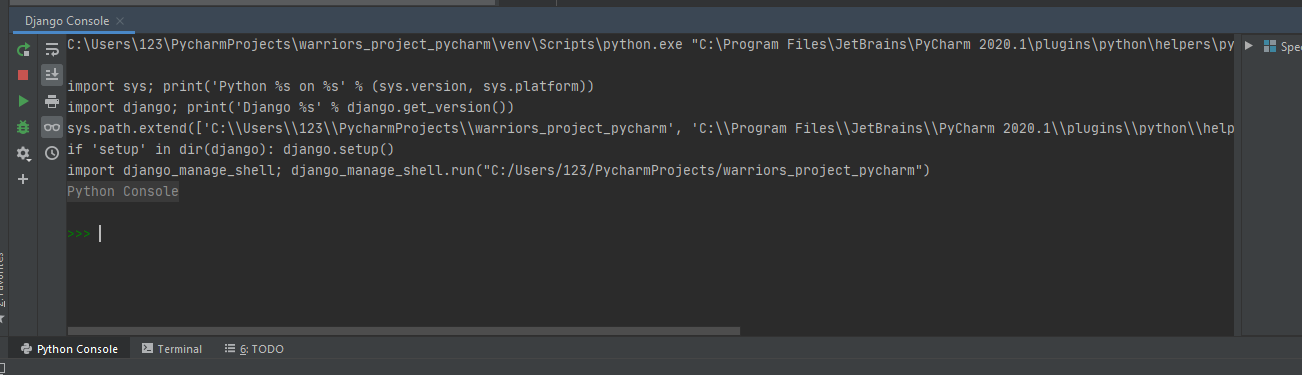
Практическое занятие №3.1

# Django Web framework. Запросы и их выполнение.

## Создание объектов

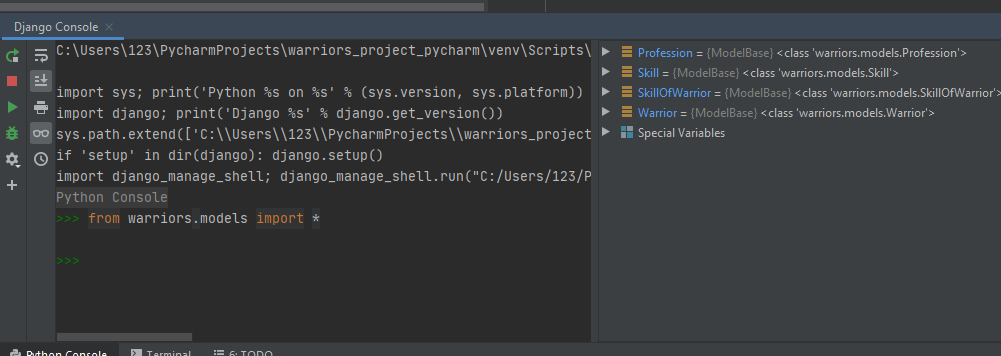
*Реализация новых объектов в Django может осуществляться не только через панель администратора - в большинстве случаев вам придется создавать новые записи в БД через внутренние методы ORM. По логике реализации подобные запросы соответствуют SQL-запросам но с учетом синтаксиса языка и фреймворка.*

### **Запуск интерактивного режима**

В дальнейшем, все задания в этой работе будут реализовываться через интерактивный интерпретатор, запущенный посредством управляющего файла manage.py. Если вы используете IDE Pycharm Professional Edition и реализовывали проект через Шаблон Django, то интерпретатор (во вкладке python) автоматически запустится через управляющий файл фреймворка:

Если же вы пользуетесь командной строкой для взаимодействия с проектом, то в папке вашей разработки для открытия интерпретатора, поддерживающего Django нужно написать  ***manage.py shell*** (для Windows) или ***python3 manage.py shell*** (для Linux):

|  |
| --- |
| (venv) C:\Users\123\PycharmProjects\warriors\_project\_pycharm>manage.py shell Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD64)] on win32 Type "help", "copyright", "credits" or "license" for more information. (InteractiveConsole) >>> |

После открытия интерпретатора, нужно импортировать классы моделей из файла models.py нашего приложения:

*Вообще, считается плохим тоном при реализации проекта на Python делать импорт через \* (из-за возможности возникновения циклических зависимостей и неиспользуемых классов переменных и методов), но т.к. мы используем интерактивный режим, то простительно.*

Теперь можно приступить к созданию наших объектов.

### **Создание объектов**

Объекты в Django можно создавать двумя способами:

1. Реализацией записи таблицы в базе как класс *(классом можно это считать условно, т.к. models.Model от которого наследуются все реализуемые нами модели использует иную логику поведения объекта на уровне [метакласса](https://habr.com/ru/company/piter/blog/592127/)).* Такой метод создания, удобен, когда нужно хранить объект в памяти некоторое время динамически преобразуя его, не отправляя дополнительных запросов к базе. Когда нужно сохранить запись, у объекта вызывается метод *[.save()](https://docs.djangoproject.com/en/3.2/ref/models/instances/" \l "saving-objects)*

Пример:

|  |
| --- |
| short\_warrior=Warrior(race="s", name="Николай Леонтьев", level=20) short\_warrior <Warrior: Николай Леонтьев> short\_warrior.save() |

*Этот же метод используется для изменения уже существующего объекта, подробней об этом в следующем пункте раздела.*

2. Реализация через менеджер *objects****.*** Любую запись можно сразу же создать в БД, если вызвать его класс через стандартный менеджер с методом [*.create()*](https://docs.djangoproject.com/en/3.2/ref/models/querysets/#django.db.models.query.QuerySet.create)и передачи в последний полей как аргументы. Результат создания объекта можно также получить как объект класса для последующий внесений изменений через *.save()*

Пример:

|  |
| --- |
| smart\_warrior = Warrior.objects.create(race="s", name="Дмитрий Мартынов", level=66) smart\_warrior <Warrior: Дмитрий Мартынов> |

### **Изменение и добавление many-to-many**

Каждый созданный объект, если вы решили его сохранить в памяти, можно впоследствии изменить. Для этого достаточно обратиться к столбцу таблицы как к полю класса и вызвать *.save()* для применений этих изменений в БД. Переведем нашего умного воина на должность тимлида:

|  |
| --- |
| smart\_warrior.race="t" smart\_warrior.save() smart\_warrior.race 't' |

Таким же можно заниматься и при реализации отношений между объектами - в нашем случае, создадим новую many-to-many связь между коротким воином и новой способностью через ассоциативную сущность SkillOfWarrior:

|  |
| --- |
| new\_skill = Skill.objects.create(title="Взлом компьютера") skill\_related = SkillOfWarrior.objects.create(skill=new\_skill, warrior=short\_warrior, level=15) skill\_related.skill <Skill: Взлом компьютера> skill\_related.warrior <Warrior: Николай Леонтьев> |

Как можно заметить, для добавления внешних таблиц использовались уже существующие объекты. Они точно также могут присваиваться к полям модели во время изменения объекта как и другие значение, по примерам выше.

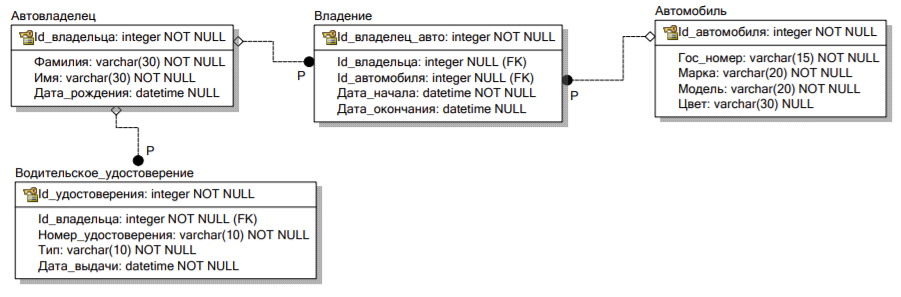
Чтобы добавить новые объекты для связи многие-ко-многим также можно к полю, реализующему данное отношение применить метод *.add()*, в аргументы которого передать все объекты, которые надо связать с нашим исходным объектом. Также, часто, может встретиться ситуация, что у нас есть некоторый итерируемый объект с записями, которые надо привязать. Тогда в метод *.add()* надо передать подобный объект указав перед ним оператор распаковки “\*”

Пример:

|  |
| --- |
| short\_warrior.skill.add(java\_skill) short\_warrior.skill <django.db.models.fields.related\_descriptors.create\_forward\_many\_to\_many\_manager.<locals>.ManyRelatedManager object at 0x0000022FBE31F358> short\_warrior.skill.all() <QuerySet [<Skill: Взлом компьютера>, <Skill: Java programming>]> |

### **Практическое задание 1:**

Воспользуйтесь проектом из практики 2.1:

Напишите запрос на создание 6-7 новых автовладельцев и 5-6 автомобилей, каждому автовладельцу назначьте удостоверение и от 1 до 3 автомобилей. Задание можете выполнить либо в интерактивном режиме интерпретатора, либо в отдельном python-файле. Результатом должны стать запросы и отображение созданных объектов. 

*Если вы добавляете автомобили владельцу через метод .add(), не забудьте заполнить также ассоциативную сущность “владение”*

## Создание простых запросов

*Любой DQL-запрос посредством ORM в Django может осуществить с помощью вызова менеджера и функций, реализующих фильтрацию или агрегацию. Получаемые объекты зависят от используемого метода, но чаще всего вы будете работать с итерируемыми объектами типа [Queryset](https://docs.djangoproject.com/en/3.2/ref/models/querysets/), предоставляемые посредством функции [filter()](https://docs.djangoproject.com/en/3.2/ref/models/querysets/" \l "filter) или объектами класса модели типа app\_name.models.YourModel, содержащиеся в Queryset’е получаемые с помощью функции [.get()](https://docs.djangoproject.com/en/3.2/ref/models/querysets/" \l "django.db.models.query.QuerySet.get).*

### **Фильтрация и получение объектов**

Для выполнения какого-либо запроса необходимо через класс модели обратиться к менеджеру объекта. [Manager](https://docs.djangoproject.com/en/3.2/topics/db/managers/) - это интерфейс, через который для моделей Django предоставляются операции запросов к базе данных. У любой модели всегда есть как минимум один менеджер - *objects,* Но при желании вы можете реализовать и [другие менеджеры](https://docs.djangoproject.com/en/3.2/topics/db/managers/)

Самым простым способом, как можно получить все объекты является метод all(). Получим таким образом всех наших воинов:

|  |
| --- |
| Warrior.objects.all() <QuerySet [<Warrior: Эдуард Ошурок>, <Warrior: Николай Леонтьев>, <Warrior: Дмитрий Мартынов>]> |

Если нужно получить список объектов по какому-то определенному условию (аналог WHERE в SQL), можно воспользоваться методом filter() позволяющий находить объекты по значениям в полях. К примеру, получим всех воинов, чей уровень равен 20:

|  |
| --- |
| Warrior.objects.filter(level=20) <QuerySet [<Warrior: Николай Леонтьев>]> |

У нас нашелся только один объект т.к база не особо заполнена, но возвращаемый объект Queryset по формату отображения намекает, что объект итерируемый и может хранить в себе более одного значения. Обратимся к 0-му элементу контейнера чтобы показать какого типа объекты хранит кварисет:

|  |
| --- |
| Warrior.objects.filter(level=20)[0] <Warrior: Николай Леонтьев> Warrior.objects.filter(level=20)[0].name 'Николай Леонтьев' |

В примере отображено, что внутри такой коллекции хранится обычный объект модели, к которой мы можем обратиться.

Можно получить сразу же конкретную запись как объект класса если воспользоваться функцией .get(). Синтаксис у нее аналогичен .filter():

|  |
| --- |
| Warrior.objects.get(level=20) <Warrior: Николай Леонтьев> |

Django выдаст ошибку, если более чем один элемент соответствует запросу get(). В этом случае он вызовет исключение MultipleObjectsReturned.

Помимо *.filter()* в Django есть функция фильтрации *.exclude().* Она производит операцию обратную фильтру и исключает все данные по запросу. Найдем таким образом всех воинов у которых уровень не равен 20:

|  |
| --- |
| Warrior.objects.exclude(level=20) <QuerySet [<Warrior: Эдуард Ошурок>, <Warrior: Дмитрий Мартынов>]> |

### **Queryset и цепочки фильтров**

Как было сказано выше Queryset - это итерируемый объект, хранящий в себе записи из БД в виде реализации объектов класса модели. К любому Queryset’у можно применять все методы описанные в пункте 1, данные методы будут применены исключительно уже к существующей выборке внутри фильтруемого объекта. Таким образом мы можем применять цепочки фильтров.

Создадим несколько новых объектов и применим подобную цепочку:

|  |
| --- |
| powerful\_warrior=Warrior.objects.create(name="Дмитрий Урбан", level="11", race="d") cute\_warrior=Warrior.objects.create(name="Никита Михайловский", level="15", race="s") Warrior.objects.exclude(level=20).filter(race="s") <QuerySet [<Warrior: Никита Михайловский>]> |

Таким образом мы получили всех воинов расы “Студент”, чей уровень не равен 20. Очевидно, что подобные цепочки к Queryset-ам можно делать до бесконечности длинные - все зависит лишь от вашей фантазии и целесообразности подобных действий.

Еще несколько полезных фактов о подобном типе:

* При применении нового метода фильтрации создается новый уникальный Queryset
* QuerySets ленивые - процесс создания QuerySet не связан с какими-либо действиями с базой данных и никаких запросов выполняться не будет без явной попытки получить объекты оттуда (например, проитерироваться)
* Аналог LIMIT в SQL для Queryset-а это обычное ограничение итерируемого объекта (прим.: *object[:5])*

### **Функции применяемые к полям и поиск через отношения**

Также как и в SQL в Django ORM есть большой набор различных критериев для фильтрации полный их список содержиться [здесь](https://docs.djangoproject.com/en/3.2/ref/models/querysets/" \l "field-lookups), мы рассмотрим лишь некоторые из них..

Чтобы применить какой-то дополнительный элемент фильтрации к полю, после его ввода необходимо дополнить значением “*\_\_function\_name*”.

Какие основные элементы поиска вам могут понадобится?

* *contains/icontains* - поиск по вхождению в строку с учетом/без учета регистра соответственно. Пример: Все объекты с именем “Дмитрий”

|  |
| --- |
| Warrior.objects.filter(name\_\_contains="Дмитрий") <QuerySet [<Warrior: Дмитрий Мартынов>, <Warrior: Дмитрий Урбан>]> |

* *in -* вхождение поля в какой-либо итерируемый объект как отдельный элемент этого объекта. Пример: Все тимлиды и студенты

|  |
| --- |
| Warrior.objects.filter(race\_\_in=["s","t"]) <QuerySet [<Warrior: Николай Леонтьев>, <Warrior: Дмитрий Мартынов>, <Warrior: Никита Михайловский>]> |

* *gt/gte/lt/lte* - Больше/Больше или равно/Меньше/Меньше или равно для определенного значения. Пример: Все воины с уровнем больше или равным 20:

|  |
| --- |
| Warrior.objects.filter(level\_\_gte=20) <QuerySet [<Warrior: Николай Леонтьев>, <Warrior: Дмитрий Мартынов>]> |

* *regex* - Применяет регулярное выражение к запросу на поиск: Пример: Имя воина содержащее буквы только с А до С

|  |
| --- |
| Warrior.objects.filter(name\_\_regex=r'([А-С])\w+ \w+') <QuerySet [<Warrior: Николай Леонтьев>, <Warrior: Дмитрий Мартынов>, <Warrior: Дмитрий Урбан>, <Warrior: Никита Михайловский>]> |

Такой синтаксис (нижнее подчеркивание после названия поля) можно применить не только к специализированным функциям для поиска - он также позволяет обращаться к полям связанных объектов, проводить по ним фильтрацию и применять функции к полям.

Создадим несколько объект умений и привяжем его к нашим воинам:

|  |
| --- |
| gaming=Skill.objects.create(title="Умение играть во все подряд") powerful\_warrior.skill.add(gaming) cute\_warrior.skill.add(gaming) |

Теперь попытаемся получить всех воинов, обладающих таким умением и имеющие 15ый уровень. Для этого отфильтруем объекты по названию умения через обращение к полю skill:

|  |
| --- |
| Warrior.objects.filter(level=15, skill\_\_title="Умение играть во все подряд") <QuerySet [<Warrior: Никита Михайловский>]> |

Подобный “проброс полей” можно производить и дальше - если бы у нас были какие-то дополнительные таблицы с внешними ключами в модели Skill, мы бы могли обратиться к полям и этой модели.

Такая логика фильтрации похожа на применение WHERE к какому-то SQL-запросу с JOIN из нескольких таблиц.

Не всегда нужно фильтровать по полям таблицы на которую ссылается модель - иногда нужно написать запрос обращаясь к объекту который, наоборот, ссылается на рассматриваемую таблицу. Специально для таких ситуаций в Django есть специальный аргумент для полей внешних ключей - *related name* он позволяет делать запрос к исходной таблице через описываемую в ссылке по имени, написанному в аргументе. Укажем подобное имя для таблицы SkillOfWarrior:

|  |
| --- |
| class SkillOfWarrior(models.Model):  """  Описание умений война  """   skill = models.ForeignKey('Skill', verbose\_name='Умение',   on\_delete=models.CASCADE)  warrior = models.ForeignKey('Warrior',   verbose\_name='Воин', related\_name="warrior\_skill",   on\_delete=models.CASCADE)  level = models.IntegerField(verbose\_name='Уровень освоения умения',   blank=True, null=True) |

**Не забудьте применить миграции!**

Чтобы выполнить следующий запрос добавим уровень созданным объектам SkillOfWarrior, которые получились у нас ранее при добавлении способностей двум новым воинам.

|  |
| --- |
| cute\_skill=SkillOfWarrior.objects.get(skill=gaming, warrior=cute\_warrior) cute\_skill.level=12 cute\_skill.save() powerful\_skill=SkillOfWarrior.objects.get(skill=gaming, warrior=powerful\_warrior) powerful\_skill.level=22 powerful\_skill.save() |

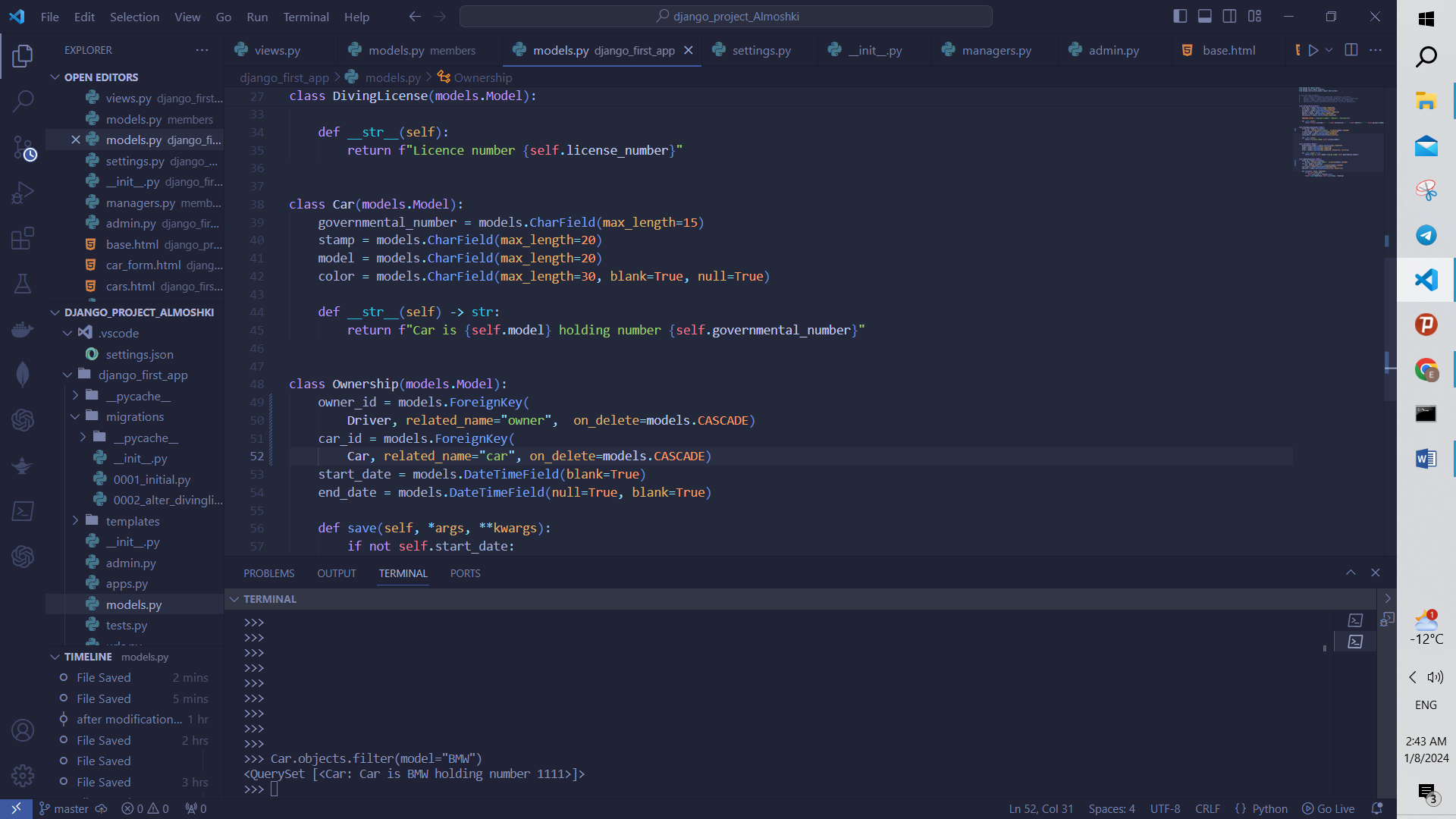
Теперь попробуем получить воина с геймерским умением такого умения больше 13:

|  |
| --- |
| Warrior.objects.filter(warrior\_skill\_\_level\_\_gt=13, skill\_\_title\_\_icontains="играть") <QuerySet [<Warrior: Дмитрий Урбан>]> |

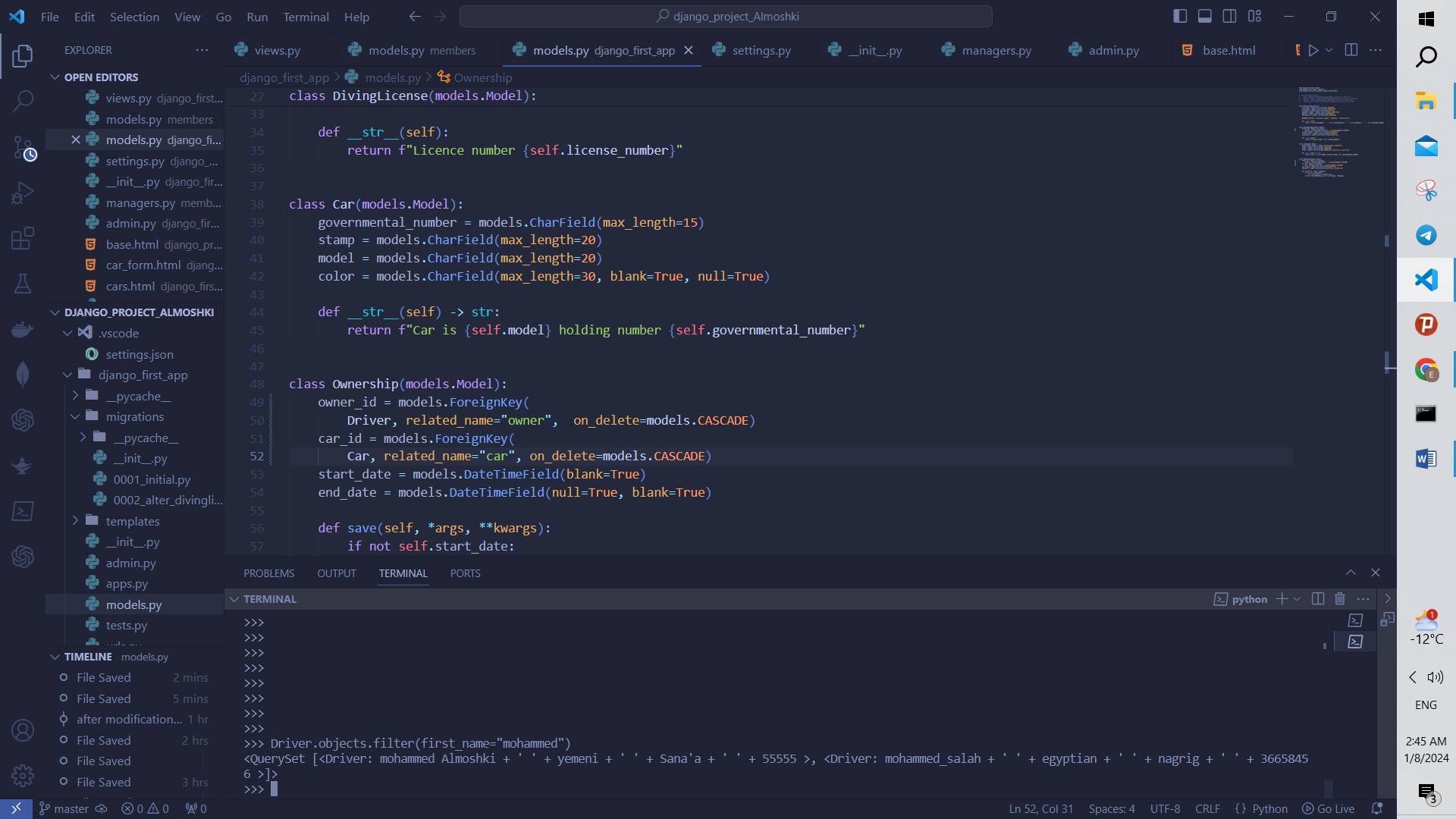
### **Практическое задание 2:**

По созданным в пр.1 данным написать следующие запросы на фильтрацию:

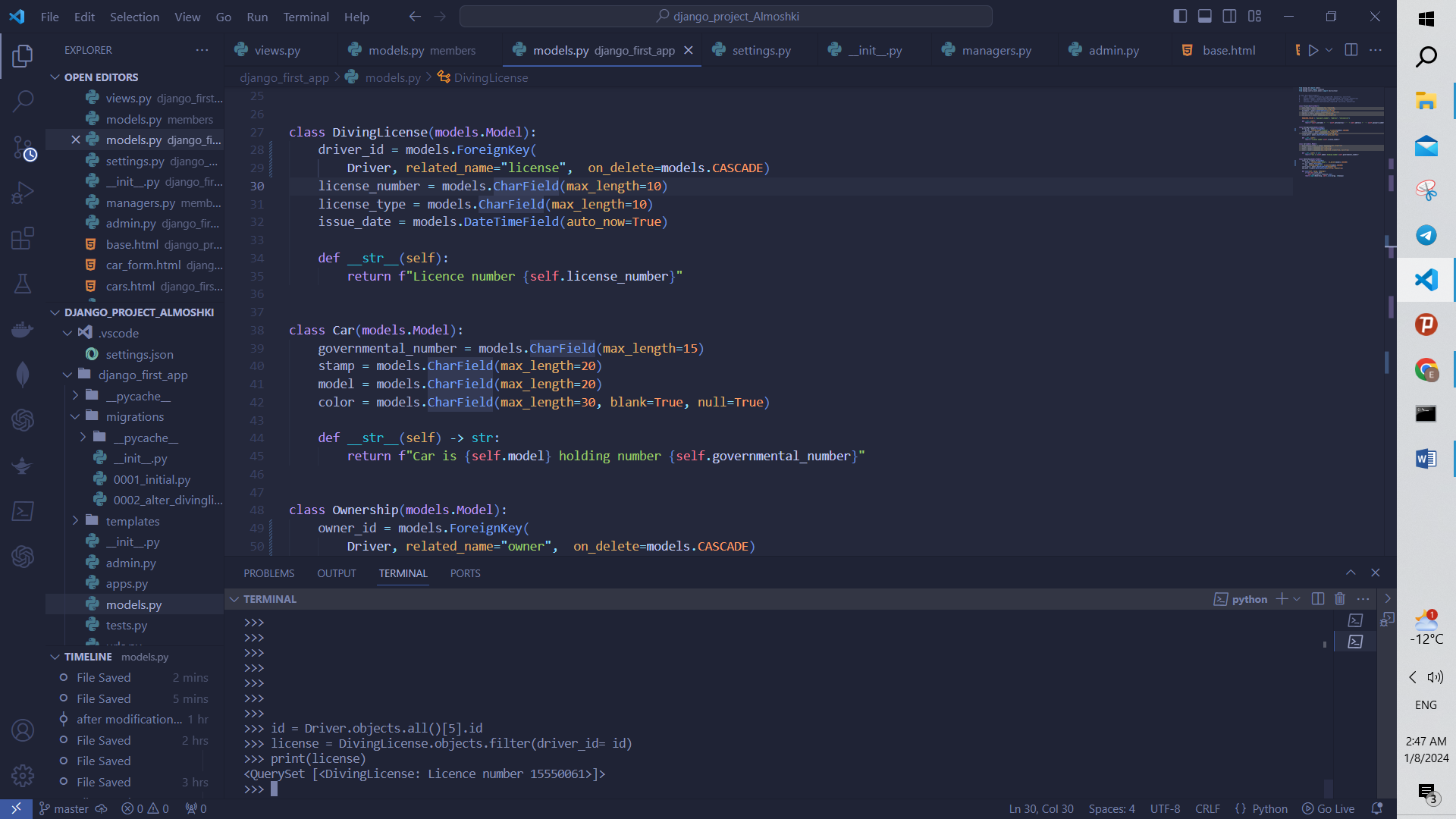
* Где это необходимо, добавьте related\_name к полям модели
* Выведете все машины марки “Toyota” (или любой другой марки, которая у вас есть)



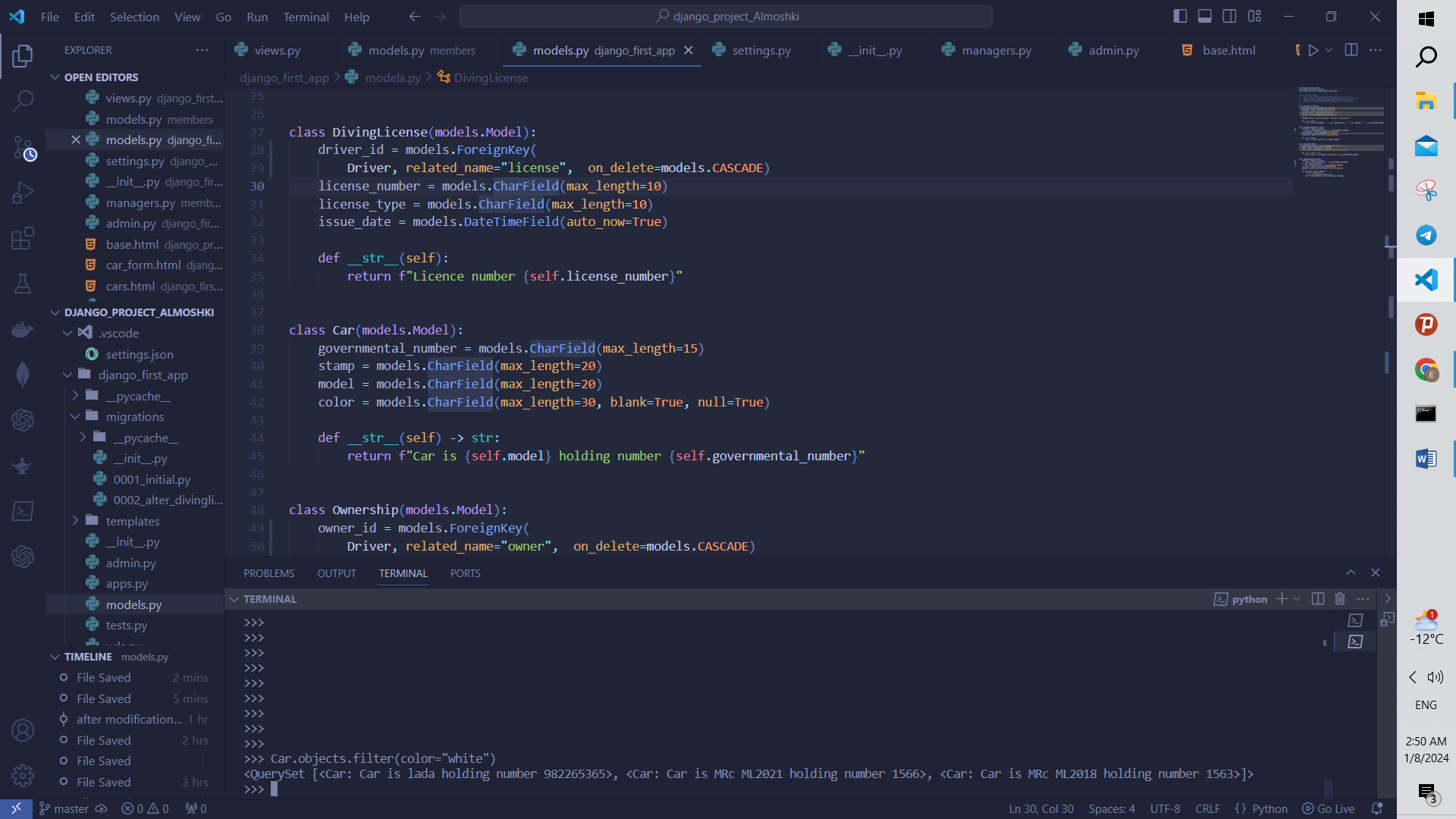
* Найти всех водителей с именем “Олег” (или любым другим именем на ваше усмотрение)

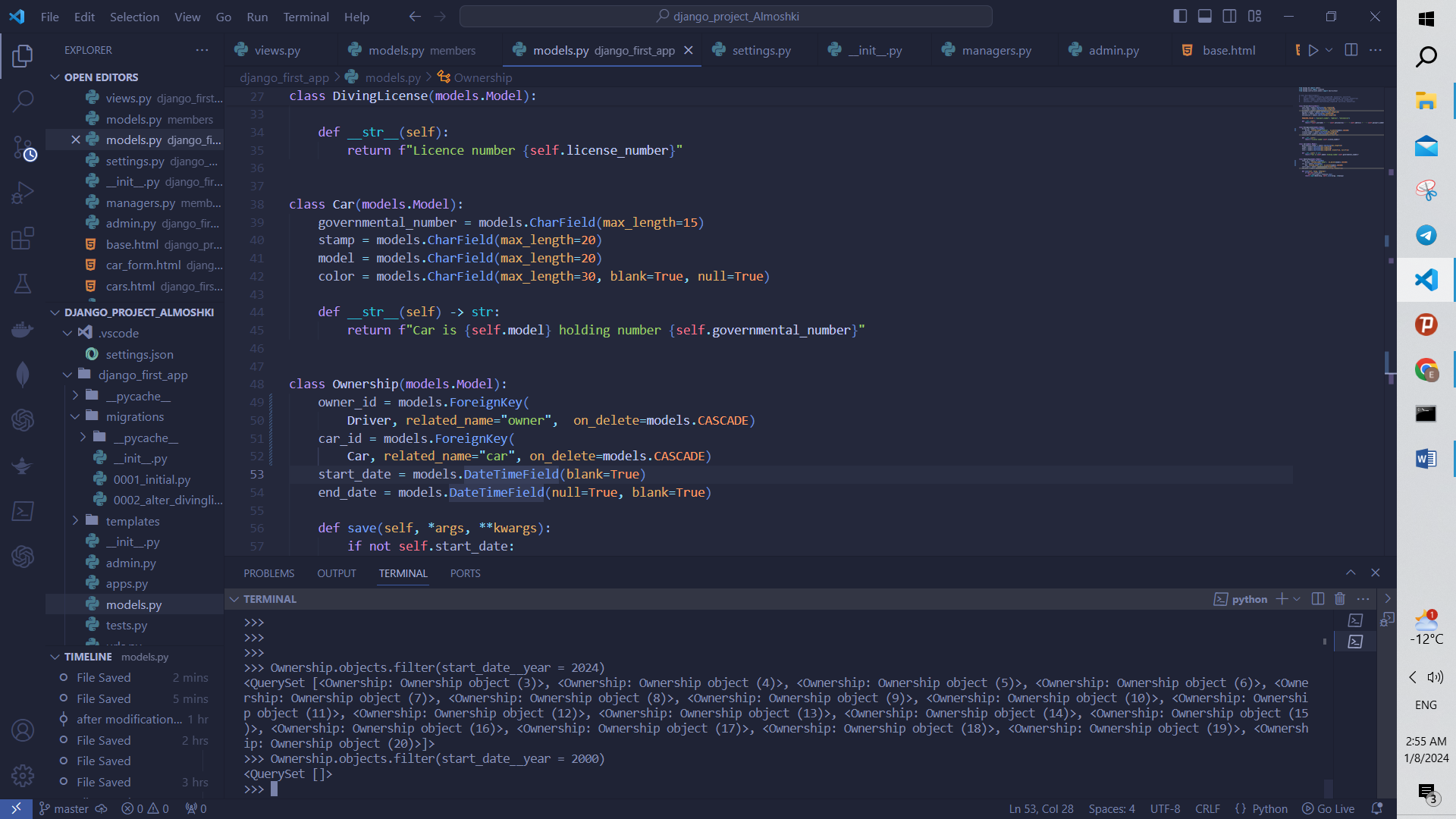


* Взяв любого случайного владельца получить его id, и по этому id получить экземпляр удостоверения в виде объекта модели (можно в 2 запроса)



* Вывести всех владельцев красных машин (или любого другого цвета, который у вас присутствует)



* Найти всех владельцев, чей [год владения машиной](https://docs.djangoproject.com/en/3.2/ref/models/querysets/" \l "year) начинается с 2010 (или любой другой год, который присутствует у вас в базе)  
    
  

## Агрегация и аннотация запросов

*Помимо основного инструментария для выполнения запросов, Django предоставляет различные возможности для нахождения информации по всему Queryset-у. С помощью функций [.annotate()](https://docs.djangoproject.com/en/3.2/ref/models/querysets/" \l "annotate) и [.aggregate()](https://docs.djangoproject.com/en/3.2/ref/models/querysets/" \l "aggregate) можно выполнить большинство запросов аналогичных запросов со встроенными функциям в SQL ( MAX, COUNT и т.д), а метод [.values()](https://docs.djangoproject.com/en/3.2/ref/models/querysets/" \l "values) позволит реализовывать аналоги запросов с GROUP BY*

### **Агрегация**

Начнем с козырей: функция *.aggregate()*, как ни странно, занимается агрегированием объектов базы данных. Ее можно применять как и к менеджеру, так и к Queryset-у, но возвращает она словарь вида:   
*{“имя функции/заданное имя”: “рассчитанное значение”}*Такой метод должен принимать в себя как минимум одну из [стандартных функций](https://docs.djangoproject.com/en/3.2/ref/models/querysets/" \l "aggregation-functions) находящихся в репертуаре Django для агрегации. Покажем на примере подсчет среднего уровня воина:

|  |
| --- |
| from django.db.models import Avg Warrior.objects.aggregate(Avg("level")) {'level\_\_avg': 22.8} |

В агрегат можно передавать несколько таких функций а самостоятельно указывать имена полей. Например, найдем минимальный и максимальный уровень воина:

|  |
| --- |
| from django.db.models import Min, Max Warrior.objects.aggregate(weakest\_powerlevel=Min("level"), highest\_powerlevel=Max("level")) {'weakest\_powerlevel': 2, 'highest\_powerlevel': 66} |

По аналогии с запросами, функции агрегирования можно вызывать и для полей объектов находящихся в отношении с моделью посредством цепочки имен любой глубины. Попробуем таким образом найти среднее значение уровня умений у воинов:

|  |
| --- |
| Warrior.objects.aggregate(Avg("warrior\_skill\_\_level")) {'warrior\_skill\_\_level\_\_avg': 16.333333333333332} |

Стоит отметить, что агрегаты как и подобные ей методы менеджеров могут взаимодействовать с Queryset. Т.е. вы всегда можете сначала можете отфильтровать выборку, а только потом проагрегировать полученный результат

### **Аннотация**

*Не путайте с аннотированием типов, это разные вещи*

Иногда нужно создавать агрегаты не для самой модели, а для таблиц, находящихся в отношении с исходной моделью. В таком случае используется функция *.annotate()* к полю внешней таблицы (или самой этой таблицы в случае Count) с указанием метода агрегации, применяемого на функцию. При таком вызове создается Queryset с объектами моделей и новым полем, реализуемым по аналогии с агрегацией. Сам метод *.annotate()* вычисляет агрегационную функцию для каждого набора записей относящихся к одному экземпляру исходной модели, группируя таблицы. Такой метод применять и полям типа *ManyToMany* и к *related\_filed*

Для следующего примера получим список всех умений и привяжем его нашим новым воинам:

|  |
| --- |
| skills=Skill.objects.all() cute\_warrior.skill.add(\*skills) powerful\_warrior.skill.add(\*skills) |

Теперь, сделаем запрос, который подсчитает конечное количество умений у каждого из воинов:

|  |
| --- |
| counter\_skills=Warrior.objects.annotate(Count("skill")) for warrior in counter\_skills: ... print(warrior.name, warrior.skill\_\_count) ...  Николай Леонтьев 2 Дмитрий Мартынов 0 Дмитрий Урбан 5 Никита Михайловский 5 Эдуард Ошурок 0 |

Как можно заметить, мы обращались с объектами Queryset-a как с обычными объектами моделей, у которых есть новое поле, сгенерированное посредством названия столбца и примененного метода.

Теперь получим минимальный уровень каждого умения у воинов:

|  |
| --- |
| min\_level\_skills=\  Warrior.objects.annotate(min\_lvl=Min("warrior\_skill\_\_level")) for warrior in min\_level\_skills: ... print(warrior.name, warrior.min\_lvl) ...  Николай Леонтьев 15 Дмитрий Мартынов None Дмитрий Урбан 22 Никита Михайловский 12 Эдуард Ошурок None |

### **Группировка посредством *.values()* и упорядочивание**

При выполнении запросов рано или поздно нужно воспользоваться старым-добрым GROUP BY. Разработчики Django почему-то решили усложнить жизнь изучающим фреймворк программистам и придумали называть метод для группировки как “*.values()”.* Этот метод обязательно вызывать вместе с *.annotate()* в позиции перед последним; *values()* принимает в качестве параметров названия полей, а возвращает словарь со значением поля и вычисленной агрегационной функцией. Таким образом, группировку можно производить не только по полям основной модели, но и по полям связанных таблиц, а аннотирование наоборот - становится валидным для полей основной таблицы.

Составим запрос с группировкой, где подсчитаем количество воинов разных рас:

|  |
| --- |
| Warrior.objects.values("race").annotate(Count("id")) <QuerySet [{'race': 'd', 'id\_\_count': 2}, {'race': 's', 'id\_\_count': 2}, {'race': 't', 'id\_\_count': 1}]> |

Помимо группировки, Django также предоставляет метод сортировки по полям. Тут почему-то решили не издеваться над названием метода и присвоили оному оригинальное имя [*.order\_by().*](https://docs.djangoproject.com/en/3.2/ref/models/querysets/#order-by)Такая функция позволяет упорядочить Queryset как по полю основной модели, так и по полям связанных таблиц. Может быть вызвана через менеджер или через созданный Queryset и возвращает все тот же, но уже упорядоченный Queryset .

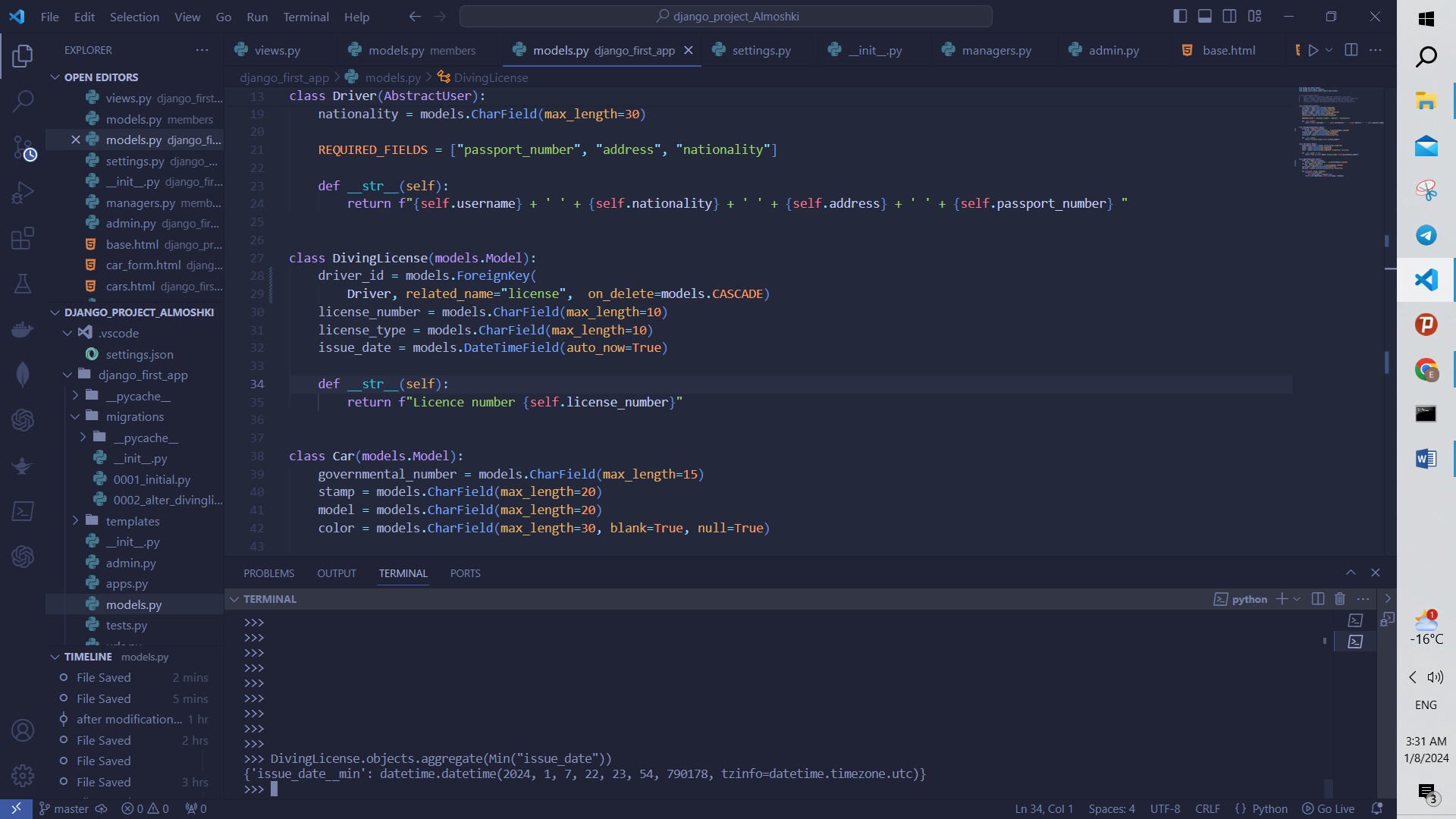
Отсортируем воинов по их уровню:

|  |
| --- |
| Warrior.objects.order\_by("level") <QuerySet [<Warrior: Эдуард Ошурок>, <Warrior: Дмитрий Урбан>, <Warrior: Никита Михайловский>, <Warrior: Николай Леонтьев>, <Warrior: Дмитрий Мартынов>]> |

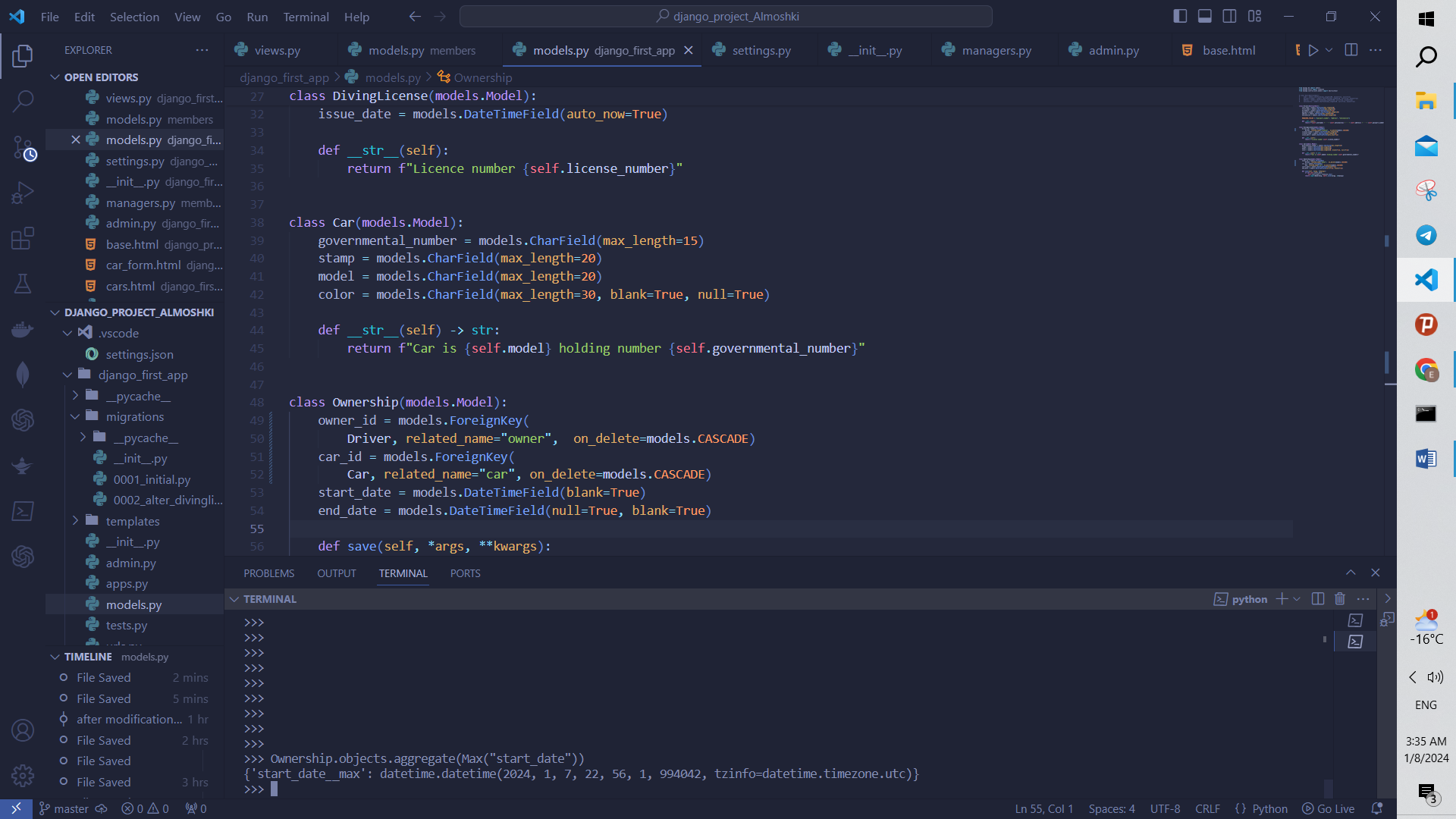
### **Практическое задание 3:**

Необходимо реализовать следующие запросы c применением описанных методов:

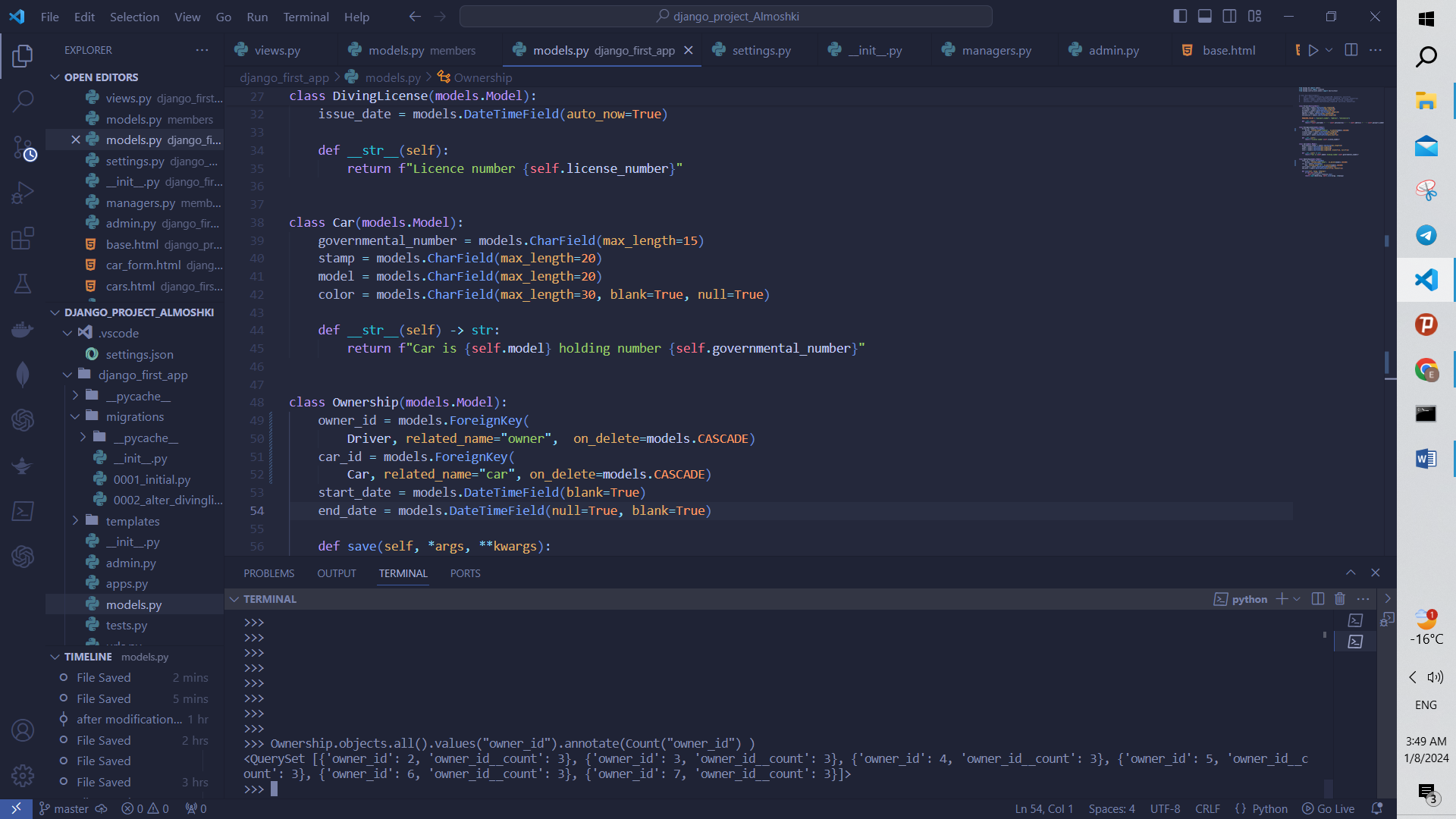
* Вывод даты выдачи самого старшего водительского удостоверения



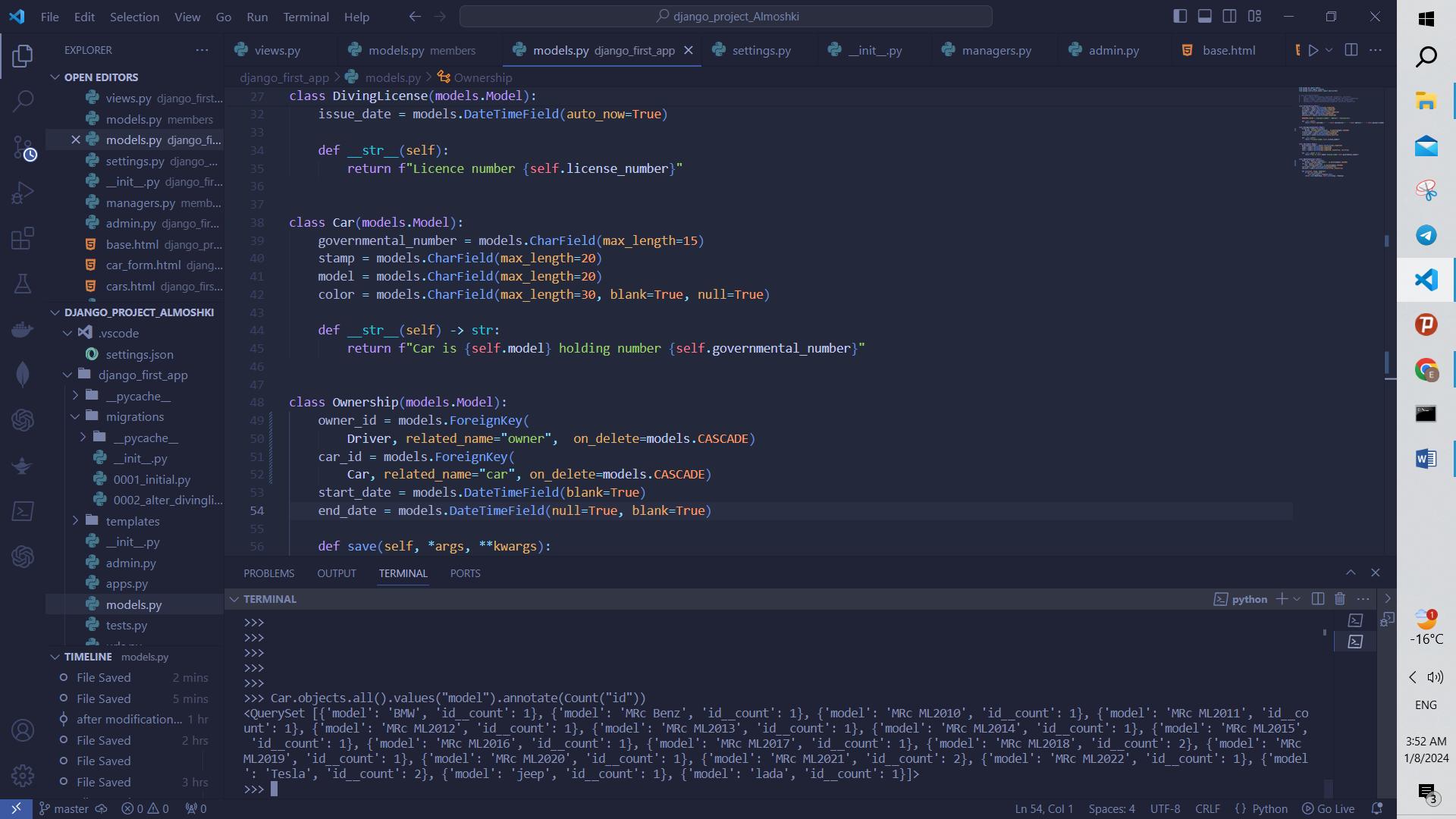
* Укажите самую позднюю дату владения машиной, имеющую какую-то из существующих моделей в вашей базе



* Выведите количество машин для каждого водителя



* Подсчитайте количество машин каждой марки



* Отсортируйте всех автовладельцев по дате выдачи удостоверения (Примечание: чтобы не выводить несколько раз одни и те же таблицы воспользуйтесь методом [.distinct()](https://docs.djangoproject.com/en/3.2/ref/models/querysets/#django.db.models.query.QuerySet.distinct)

