**Тема занятия № 5: Модели. Базовые инструменты. Миграции**

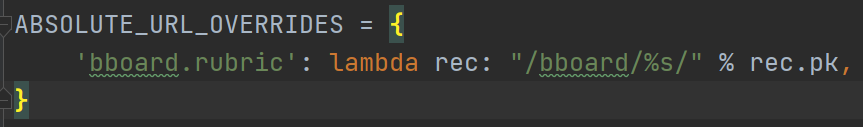
1. Интернет-адрес модели и его формирование

Django позволяет сформировать интернет-адрес, указывающий на конкретную запись модели, — интернет-адрес модели. При переходе по нему может открываться страница с содержимым этой записи, списком связанных записей и др. Сформировать интернет-адрес модели можно двумя способами: декларативным и императивным.

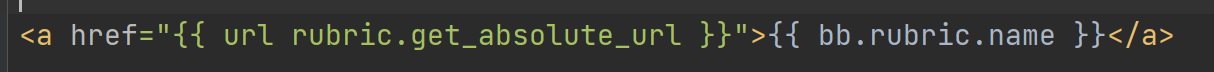
Декларативный способ заключается в описании формата интернет-адреса в настройках проекта. Набор таких адресов оформляется в виде словаря Python и записывается в параметре absolute\_url\_overrides модуля settings.ру пакета конфигурации.

Ключи элементов ЭТОГО словаря ДОЛЖНЫ иметь ВИД <псевдоним приложения:». <имя Класса модели>. Значениями элементов станут функции, в качестве единственного параметра принимающие объект записи модели и возвращающие строку с готовым интернет-адресом. Здесь удобно использовать лямбда-функции Python.

Вот пример объявления словаря, который на основе рубрики (экземпляра класса модели Rubric) сформирует адрес вида bboard/<K/i/04 рубрики>/, ведущий на страницу со списком объявлений, относящихся к этой рубрике:

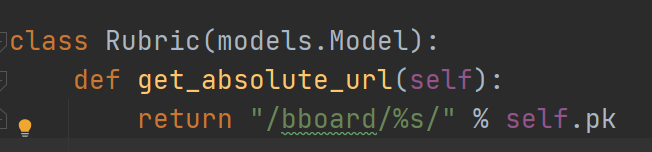


Теперь, чтобы поместить в код шаблона интернет-адрес модели, достаточно вставить туда вызов метода get\_absoiute\_uri(), унаследованного всеми моделями от базового класса Model:



Точно таким же образом можно получить интернет-адрес модели где-либо еще —например, в коде контроллера.

Императивный способ заключается в непосредственном переопределении метода Get absolute url (self) в классе модели. Вот пример:

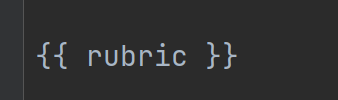


Разумеется, в параметре absolute url overrides настроек проекта в таком случае нет нужды.

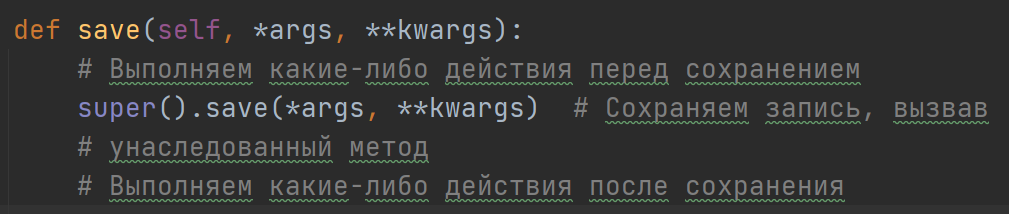
1. Методы модели

Помимо атрибутов класса, представляющих поля модели, и вложенного класса Meta, где объявляются параметры модели, в классе модели можно объявить дополнительные методы:

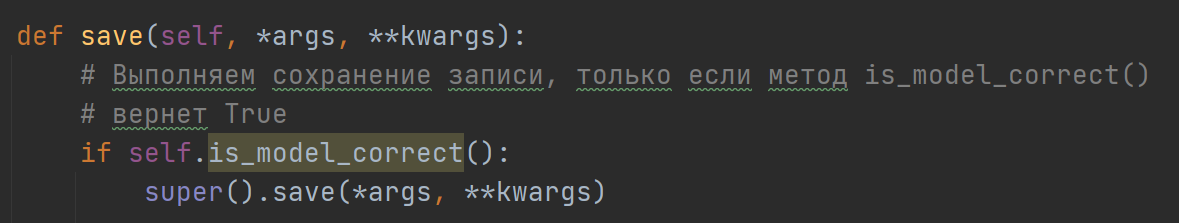
□ \_\_str\_\_(self) — возвращает строковое представление записи модели. Оно будет выводиться, если в коде шаблона указать вывод непосредственно объекта записи, а не значения его поля или результата, возвращенного его методом:



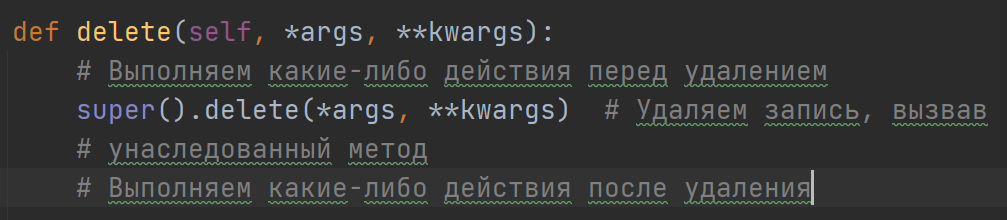
□ save (self, \*args, \*\*kwargs) — сохраняет запись. При определении этого метода обязательно следует вставить в нужное место кода вызов метода, унаследованного от базового класса. Вот пример:



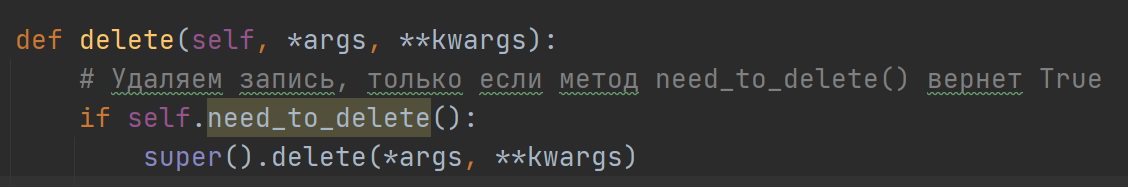
В зависимости от выполнения или невыполнения какого-то условия, можно отменить сохранение записи, для чего достаточно просто не вызывать унаследованный метод save (). Пример:



□ delete (seif, \*args, \*\*kwargs) — удаляет запись. Этот метод также переопределяется для добавления какой-либо логики, которая должна выполняться перед удалением и (или) после него. Пример:

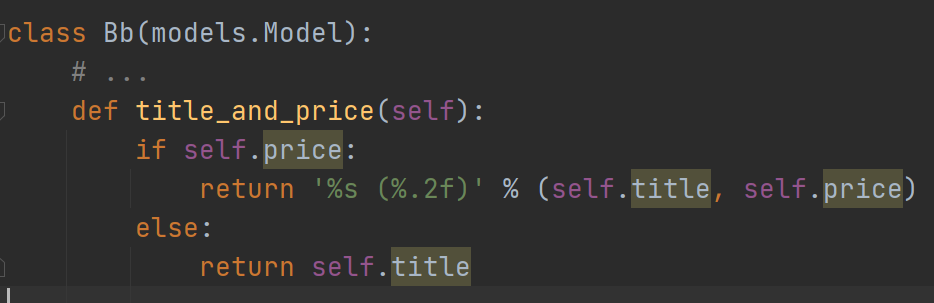


И точно таким же образом в случае необходимости можно предотвратить удаление записи:

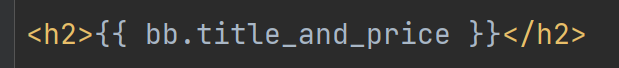


В модели можно объявить дополнительное поле, значение которого вычисляется на основе каких-то других данных и которое доступно только для чтения (функциональное поле). Для этого достаточно объявить метод, не принимающий параметров и возвращающий нужное значение. Имя этого метода станет именем функционального поля.

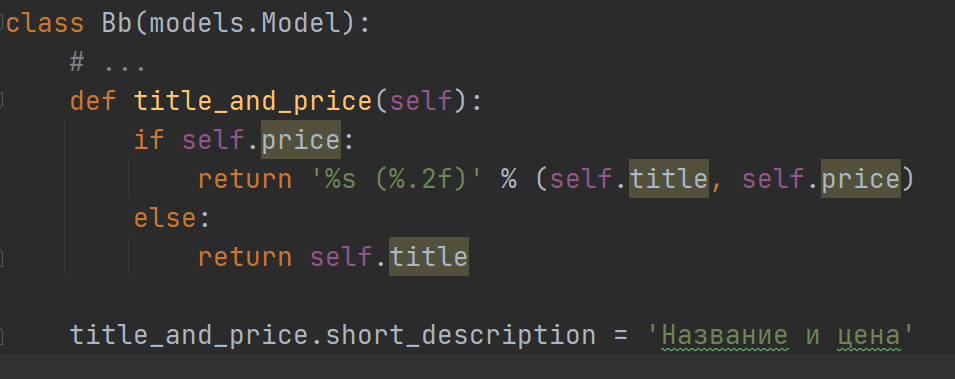
В качестве примера создадим в модели вь функциональное поле titie and price, объявив одноименный метод:



Вот так значение функционального поля выводится в шаблоне:



Для функционального поля допускается указать название, которое будет выводиться на веб-страницах. Строку с названием нужно присвоить атрибуту short\_Description объекта метода, который реализует это поле:

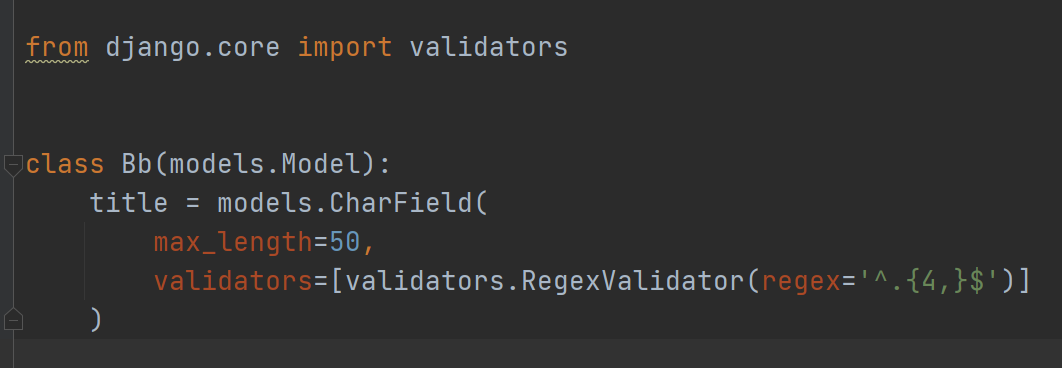


1. Валидация модели. Валидаторы

Валидацией называется проверка на корректность данных, занесенных в поля модели. Валидацию можно реализовать непосредственно в модели или же в форме, которая используется для занесения в нее данных.

Валидацию значений, заносимых в отдельные поля модели, выполняют валидаторы. Реализованные в виде функций или классов. Некоторые типы полей уже используют определенные валидаторы — так, строковое поле charfield задействует валидатор maxlengthvaiidator, проверяющий, не превышает ли длина заносимого строкового значения указанную максимальную длину.

Помимо этого, можно указать для любого поля другие валидаторы, предоставляемые django. Реализующие ИХ классы объявлены В модуле django.core.validators. А указываются они в параметре validators конструктора класса поля. Пример:



Здесь использован валидатор, представляемый классом regexvalidator. Он проверяет заносимое в поле значение на соответствие заданному регулярному выражению. Если значение не проходит проверку валидатором, он возбуждает исключение Validationerror ИЗ модуля django.core.exceptions.

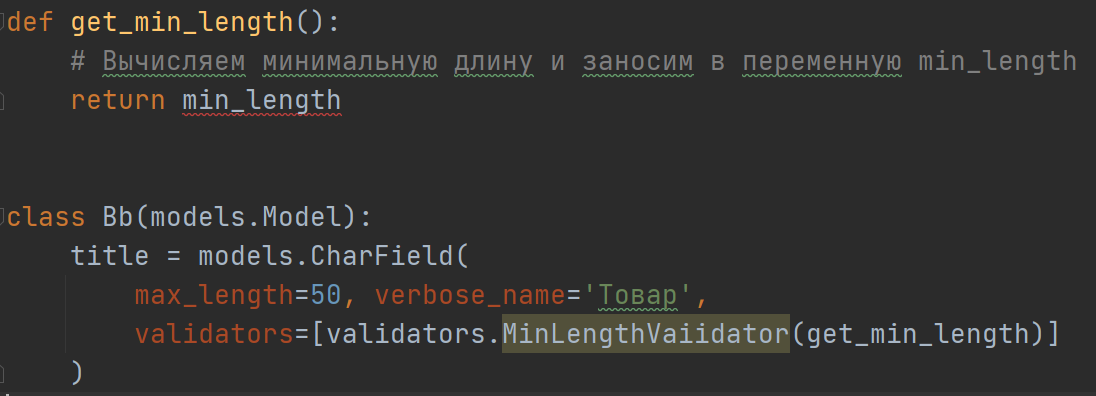
В составе Django поставляются следующие классы валидаторов:

□ minlengthvaiidator— проверяет, не меньше ли длина заносимой строки, чем минимум, заданный в первом параметре. Формат конструктора:

Minlengthvaiidator(<минимальная длина>[, message=None])

Параметр message задает сообщение об ошибке — если он не указан, то выводится стандартное сообщение. Код ошибки: "min iength".

Начиная с Django 2.2, в качестве первого параметра конструктора можно указать функцию, не принимающую параметров и возвращающую минимальную длину значения в виде целого числа. Пример:



□ maxlengthvaiidator — проверяет, не превышает ли длина заносимой строки заданный в первом параметре максимум. Используется полем типа charfield.

Формат конструктора:

Maxlengthvaiidator (<максимальная длина> [, message=None]) параметр message задает сообщение об ошибке — если он не указан, используется стандартное. Код ошибки: "max length".

Начиная с Django 2.2, в качестве первого параметра конструктора можно указать функцию, не принимающую параметров и возвращающую максимальную длину значения в виде целого числа;

□ Regexvalidator— проверяет значение на соответствие заданному регулярному выражению. Конструктор класса:

Regexvalidator(regex=None[r message=None][, code=None][,

Inverse\_match=None][, flags=0])

Он принимает следующие параметры:

• regex — само регулярное выражение. Может быть указано в виде строки или объекта типа regex, встроенного в Python;

• message — строка с сообщением об ошибке. Если параметр не указан, то выдается стандартное сообщение;

• code— код ошибки. Если не указан, используется код по умолчанию:

"invalid”;

• inverse match— если False, значение должно соответствовать регулярному выражению (поведение по умолчанию). Если True, то значение, напротив, не должно соответствовать регулярному выражению;

• flag— флаги регулярного выражения. Используется, только если таковое задано в виде строки;

□ Emailvalidator — проверяет на корректность заносимый в поле адрес электронной почты. Используется полем типа emailfieid. Конструктор класса:

Emailvalidator([message=None][,][code=None][,][, whitelist=None])

Параметры:

• message— строка с сообщением об ошибке. Если не указан, то выводится стандартное сообщение;

• code— код ошибки. Если не указан, используется код по умолчанию:

"invalid";

• whitelist — последовательность доменов (представленных в виде строк), которые не будут проверяться валидатором. Если параметр не указан, то в эту последовательность входит только адрес локального хоста localhost;

□ urlvaiidator— проверяет на корректность заносимый в поле интернет-адрес.

Используется полем типа urlfieid. Конструктор класса:

Urlvaiidator([schemes=None][,][regex=None][,][message=None][,]

[code=None])

Параметры:

• schemes — последовательность обозначений протоколов, в отношении которых будет выполняться валидация, в виде строк. Если параметр не указан, то используется последовательность [ ’ http ’, ' https’, ' ftp1, 1 ftps' ];

• regex — регулярное выражение, с которым должен совпадать интернет-адрес.

Может быть указано в виде строки или объекта типа regex, встроенного в Python. Если отсутствует, то такого рода проверка не проводится;

• message— строка с сообщением об ошибке. Если не указан, то выводится стандартное сообщение;

• code— код ошибки. Если не указан, используется код по умолчанию:

"invalid”;

□ prohibitnuiicharactersvaiidator —проверяет, не содержит ли заносимая строка нулевой символ: \хоо. Формат конструктора:

Prohibitnullcharactersvalidator([message=None][,][code=None]) он принимает следующие параметры:

• message— строка с сообщением об ошибке. Если не указан, то выдается стандартное сообщение;

• code— код ошибки. Если не указан, используется код по умолчанию:

"null\_characters\_not\_allowed";

□ minvaiuevaiidator — проверяет, не меньше ли заносимое число заданного в первом параметре минимума. Формат конструктора:

Minvaiuevaiidator(<минимальное значение>[, message=None])

Параметр message задает сообщение об ошибке; если он не указан, выводится стандартное. Код ошибки: "min vaiue".

Начиная с Django 2.2, в качестве первого параметра конструктора можно указать функцию, не принимающую параметров и возвращающую минимальное значение в виде целого числа;

□ Maxvaluevalidator— проверяет, не превышает ли заносимое число заданный в первом параметре максимум. Формат конструктора:

Maxvaluevalidator(<максимальное значение>[, message=None])

Параметр message задает сообщение об ошибке — если он не указан, выдается стандартное. Код ошибки: "max value".

Начиная с Django 2.2, в качестве первого параметра конструктора можно указать функцию, не принимающую параметров и возвращающую максимальное значение в виде целого числа;

□ Decimalvalidator— проверяет заносимое вещественное число фиксированной точности, представленное объектом типа Decimal ИЗ модуля decimal Python.

Формат конструктора:

Decimalvalidator (<максимальное количество цифр в числе>,<количество цифр в дробной части>)

Коды ошибок:

• "max digits" —если общее количество цифр в числе больше заданного;

• "max decimal places"— если количество цифр в дробной части больше заданного;

• "max whoie digits” — если количество цифр в целой части числа больше разности между общим количеством и количеством цифр в дробной части.

Часть валидаторов реализована в виде функций:

□ validate ipv4 6 address о — проверяет на корректность интернет-адреса протоколов ipv4 и ipv6;

□ vaiidate\_ipv4\_address о — проверяет на корректность интернет-адреса только протокола ipv4;

□ vaiidate\_ipv6\_address о — проверяет на корректность интернет-адреса только протокола ipv6.

Эти три валидатора используются полем типа genericipaddressfield;

□ int iist vaiidator о — возвращает экземпляр класса regexvalidator, настроенный на проверку последовательностей целых чисел, которые разделяются указанным символом-разделителем. Формат вызова:

Int\_list\_validator([sep=’,'][,][message=None][,][code=\*invalid’][,]

[allow\_negative=False])

Параметры:

• sep — строка с символом-разделителем. Если не указан, то в качестве разделителя используется запятая;

• message— строка с сообщением об ошибке. Если не указан, то выдается стандартное сообщение;

• code— код ошибки. Если не указан, используется код по умолчанию:

"invalid”;

• allow negative —если True, допускаются отрицательные числа, если False —не допускаются (поведение по умолчанию).

Помимо классов и функций, модуль django.core.validators объявляет ряд переменных. Каждая из них хранит готовый объект валидатора, настроенный для определенного применения:

□ validate email— экземпляр класса emailvalidator С настройками ПО умолчанию;

□ vaiidate slug— экземпляр класса regexvalidator, настроенный на проверку слагов. Допускает наличие в слагах только латинских букв, цифр, символов ’’Минус” и подчеркивания;

□ validate unicode slug — экземпляр класса regexvalidator, настроенный на проверку слагов. Допускает наличие в слагах только букв в кодировке Unicode, цифр, символов ’’минус” и подчеркивания;

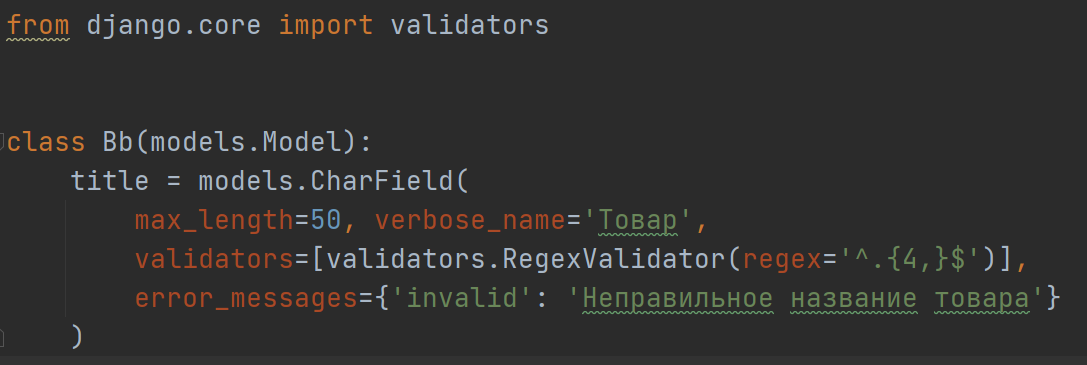
Эти два валидатора используются полем типа siugfieid;

О validate\_comma\_separated\_integer\_list — экземпляр класса regexvalidator, настроенный на проверку последовательностей целых чисел, которые разделены запятыми.

Во многих случаях стандартные сообщения об ошибках, выводимые валидаторами, вполне понятны. Но временами возникает необходимость вывести посетителю сообщение, более подходящее ситуации.

Собственные сообщения об ошибках указываются в параметре error messages конструктора класса поля. Значением этого параметра должен быть словарь Python, у которого ключи элементов должны совпадать с кодами ошибок, а значения —задавать сами тексты сообщений.

Вот пример указания для поля title модели вь собственного сообщения об ошибке:



Доступные для указания коды ошибок:

□ "null” — поле таблицы не может хранить значение null, т. Е. Его следует заполнить;

□ "blank” —в элемент управления должно быть занесено значение;

□ "invalid" —неверный формат значения;

□ "invalid choice’’ —в поле со списком заносится значение, не указанное в списке;

□ "unique" —в поле заносится неуникальное значение, что недопустимо;

□ "unique for date" — в поле заносится значение, неуникальное в пределах даты, что недопустимо;

□ "invalid date" — значение даты хоть и введено правильно, но некорректное (например, 35.14.2020);

□ "invalid time" — значение времени хоть и введено правильно, но некорректное (например, 25:73:80);

□ "invalid datetime" — значение даты и времени хоть и введено правильно, но некорректное (например, 35.14.2020 25:73:80);

□ "min iength" — длина сохраняемой в поле строки меньше указанного минимума;

□ "max length" — длина сохраняемой в поле строки больше указанного максимума;

□ "nuii characters not aiiowed" — сохраняемая строка содержит нулевые символы \хоо;

□ ’’min vaiue” — сохраняемое в поле число меньше указанного минимума;

□ "max value” — сохраняемое в поле число больше указанного максимума;

□ "max digits"— общее количество Цифр В Сохраняемом числе типа Decimal больше заданного;

□ "max decimal places”— количество цифр в дробной части сохраняемого числа типа Decimal больше заданного;

□ "max whole digits" — количество цифр в целой части сохраняемого числа типа

Decimal больше разности между максимальным общим количеством цифр и количеством цифр в дробной части.

Если нужный валидатор отсутствует в стандартном наборе, мы можем написать его самостоятельно, реализовав его в виде функции или класса.

Валидатор, выполненный в виде функции, должен принимать один параметр —значение, которое следует проверить. Если значение некорректно, то функция должна возбудить исключение validationerror ИЗ МОДУЛЯ django. Core. Exceptions.

Возвращать результат она не должна.

Для вызова конструктора класса исключения validationerror предусмотрен следующий формат:

Validationerror (<описание ошибки>[, code=None] [, params=None])

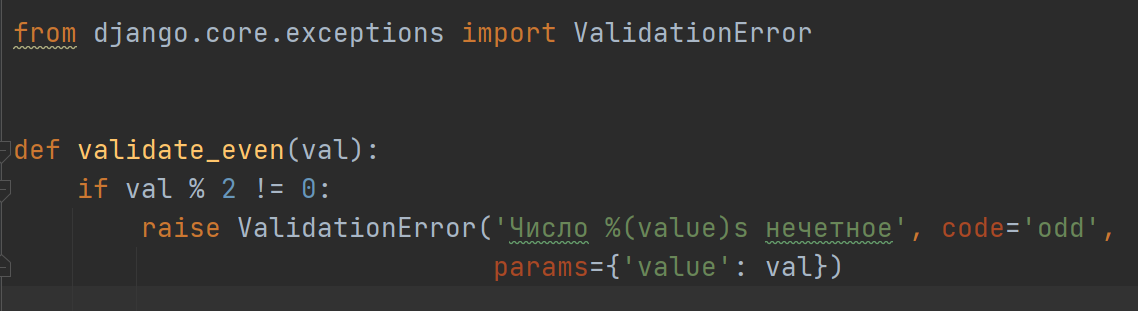
Первым, позиционным, параметром передается строка с текстовым описанием ошибки, допущенной посетителем при вводе значения. Если в этот текст нужно Вставить какое-либо значение, следует использовать заменитель вида % (<ключ элемента словаря, переданного параметром params>s).

В параметре code указывается код ошибки.

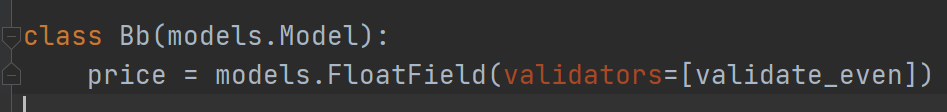
Разработчики Django настоятельно рекомендуют задавать код ошибки. Однако в этом случае нужно помнить, что текст сообщения об ошибке, для которой был указан код, может быть изменен формой, привязанной к этой модели, или самим разработчиком посредством параметра error messages конструктора поля. Поэтому, если вы хотите, чтобы заданный вами в валидаторе текст сообщения об ошибке всегда выводился как есть, не указывайте для ошибки код.

В параметре params задается словарь со значениями, которые нужно поместить в текст сообщения об ошибке (он передается первым параметром) вместо заменителей.

Далее показывает код валидатора, реализованного в виде функции vaiidate\_Even (). Он проверяет, является ли число четным.

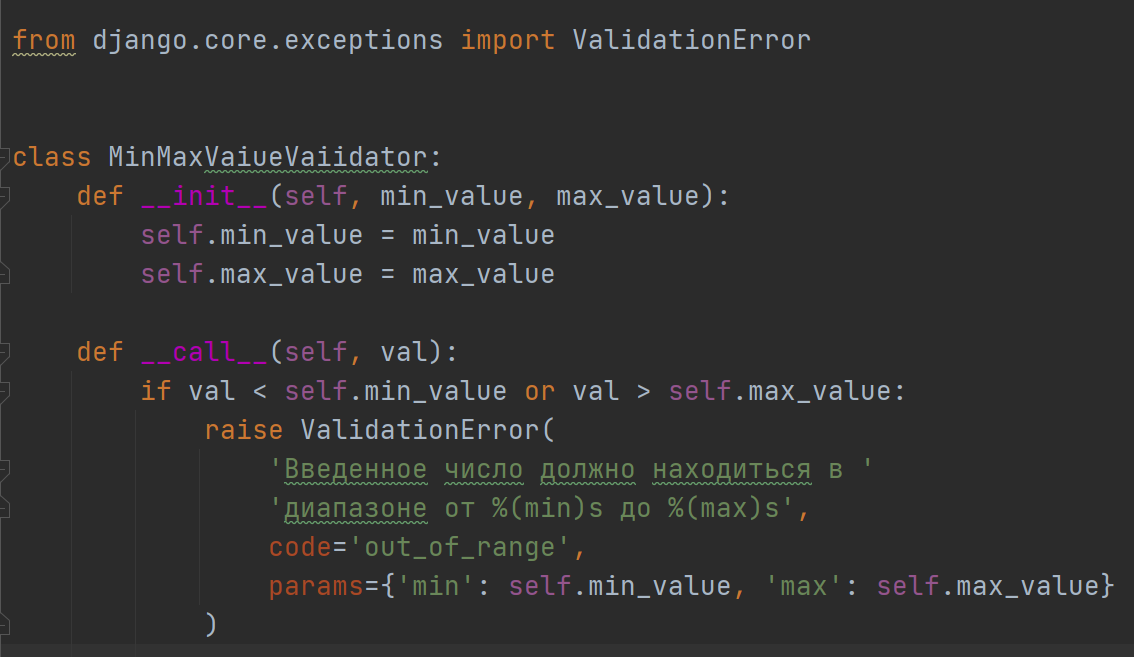


Этот валидатор указывается для поля точно так же, как и стандартный:



Если валидатору при создании следует передавать какие-либо параметры, задающие режим его работы, то этот валидатор нужно реализовать в виде класса. Параметры валидатору будут передаваться через конструктор класса, а сама валидация станет выполняться в переопределенном методе \_\_call\_\_(). Последний должен принимать с параметром проверяемое значение и возбуждать исключение Validationerror, если оно окажется некорректным.

Далее демонстрирует код класса minmaxvaiuevaiidator, проверяющего, находится ли заносимое в поле числовое значение в заданном диапазоне. Нижняя и верхняя границы этого диапазона передаются через параметры конструктора Класса.



Может возникнуть необходимость проверить на корректность не значение одного поля, а всю модель (выполнить валидацию модели}. Для этого достаточно переопределить В Классе модели метод clean (self).

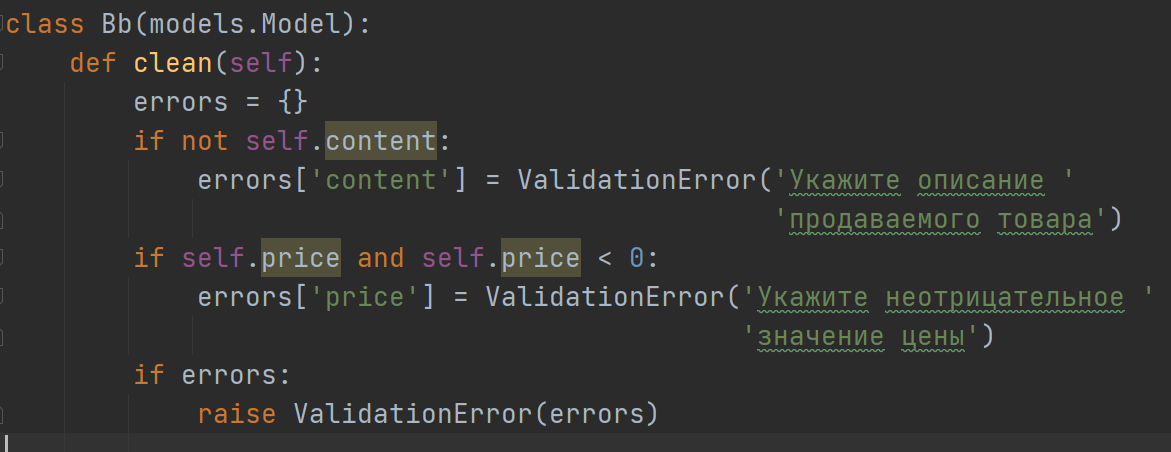
Метод не должен принимать параметры и возвращать результат. Единственное, что ОН обязан Сделать, — В Случае Необходимости возбудить исключение validationerror. Поскольку некорректные значения могут быть занесены сразу в несколько полей, валидатор должен формировать список ошибок. В этом случае пригодится второй формат конструктора класса validationerror:

Validationerror(<список ошибок>)

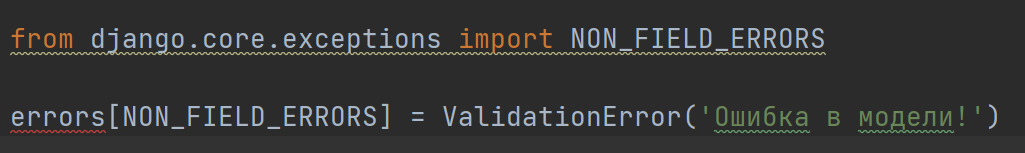
Список ошибок удобнее всего представлять в виде словаря. В качестве ключей элементов указываются имена полей модели, в которые были занесены некорректные значения. В качестве значений этих элементов должны выступать последовательности из экземпляров класса validationerror, каждый из которых представляет одну из ошибок.

Для примера сделаем так, чтобы занесение описания продаваемого товара было обязательным, и предотвратим ввод отрицательного значения цены. Вот код метода Clean (), который реализует всё это:

**Class Bb(models.Model):**

****

Если нужно вывести какое-либо сообщение об ошибке, относящейся не к определенному полю модели, а ко всей модели, то следует использовать в качестве ключа словаря, хранящего список ошибок, значение переменной non field errors из модуля dj ango. Core. Exceptions. Пример:



Конструктор класса validationerror поддерживает и прием списка ошибок в виде собственно списка или кортежа Python, однако в этом случае мы не сможем указать Django, к какому полю модели относятся те или иные ошибки, и форма, связанная с моделью, не сможет вывести их напротив соответствующего элемента управления.

1. Генерирование миграций

Для генерирования миграций служит команда makemigrations утилиты manage.py:

Manage.py makemigrations [<список псевдонимов приложении, разделенных & пробелами>] [—name | -п <имя миграции>] [—noinput] [—no-input]

[—dry-run] [—check] [—merge] [—empty] [—no-header]

****

Если псевдонимы приложений не указаны, то будут обработаны модели из всех приложений проекта. Если указать псевдоним приложения или несколько псевдонимов, разделив их пробелами, будут обработаны только миграции из указанных приложений.

Команда makemigrations поддерживает следующие ключи:

□ —name или -п — указывает имя формируемой миграции, которое будет добавлено к порядковому номеру для получения полного имени файла с кодом миграции. Если оно отсутствует, то миграция получит имя по умолчанию;

□ —noinput или -no-input —отключает вывод на экран сведений о формируемой Миграции;

□ -dry-run— выводит на экран сведения о формируемой миграции, но не формирует ее;

□ —check— выводит сведения о том, изменились ли модели после последнего формирования миграций, но не формирует саму миграцию;

□ —merge —используется для устранения конфликтов между миграциями;

□ —empty— создает ’’пустую” миграцию для программирования ее вручную.

’’Пустая” миграция может пригодиться, например, для добавления какого-либо расширения в базу данных postgresql.

□ -no-header (начиная с Django 2.2) — указывает не вставлять в начало модуля Миграции комментарий с указанием версии Django и временной отметки генерирования миграции.

Результатом выполнения команды станет единственный файл миграции, выполняющий над базой данных все необходимые действия: создание, изменение и удаление таблиц, полей, индексов, связей и правил.

Внимание!

Django отслеживает любые изменения в коде моделей, даже те, которые не затрагивают структуры базы данных напрямую. Так, если мы укажем для поля название (параметр verbose name конструктора поля), фреймворк все равно создаст в миграции код, который изменит параметры нижележащего поля таблицы в базе данных. Поэтому крайне желательно продумывать структуру моделей заранее и впоследствии, по возможности, не менять ее.

Для отслеживания, какие миграции уже были выполнены, а какие — еще нет, Django создает в базе данных по умолчанию таблицу djangojnigrations. Править вручную как её структуру, так и хранящиеся в ней записи настоятельно не рекомендуется.

1. Файлы миграций

Все программные модули миграций сохраняются в пакете migrations, расположенном в пакете приложения. По умолчанию они получают имена формата последовательно увеличивающиеся порядковые номера>\_<имя миграции>.ру. Порядковые номера состоят из четырех цифр и просто помечают очередность, в которой формировались миграции. Имя миграции задается в ключе —name (-п) команды Makemigretions— если этот ключ не указан, то фреймворк сам сгенерирует имя следующего вида:

□ initial — если это начальная миграция, т. Е. Создающая самые первые версии всех необходимых структур в базе данных;

□ а\Ло\_<отметка даты и времени формирования миграции>— если это миграции, сформированные после начальной и дополняющие, удаляющие или изменяющие созданные ранее структуры.

Миграции можно переименовывать, но только в том случае, если они до этого ещё ни разу не выполнялись. Дело в том, что имена модулей миграций сохраняются в таблице django\_migrations, и если мы переименуем уже выполненную миграцию, то Django не сможет проверить, была ли она уже выполнена, и выполнит ее снова.

1. Выполнение миграций

Выполнение миграций запускается командой migrate утилиты manage.py:

Manage.py migrate [<псевдоним приложения> [<имя миграции>]]

[—fake-initial] [—noinput] [—no-input] [—fake] [—plan]



Если не указать ни псевдоним приложения, ни имя миграции, то будут выполнены все не выполненные к настоящему моменту миграции во всех приложениях проекта.

Если указать только псевдоним приложения, будут выполнены все миграции в этом приложении, а если дополнительно задать имя миграции, то будет выполнена только эта миграция.

Задавать имя модуля миграции полностью нет необходимости — достаточно записать только порядковый номер, находящийся в начале ее имени:

Manage.py migrate bboard 0001

В команде можно применить такие дополнительные ключи:

□ -fake-initial— пропускает выполнение начальной миграции. Применяется,

Если в базе данных на момент первого выполнения миграций уже присутствуют все необходимые структуры, и их нужно просто модифицировать;

□ —noinput или -no-input — отключает вывод на экран сведений о применении Миграций;

□ —fake — помечает миграции как выполненные, но не вносит никаких изменений в базу данных. Может пригодиться, если все необходимые изменения в базу были внесены вручную;

□ —plan (начиная с Django 2.2)— выводит план миграций— список, перечисляющий миграции в порядке их выполнения.

На каждую выполненную миграцию в таблицу django\_migrations базы данных добавляется отдельная запись, хранящая имя модуля миграции, имя приложения, в котором она была создана, дату и время выполнения миграции.

1. Слияние миграций

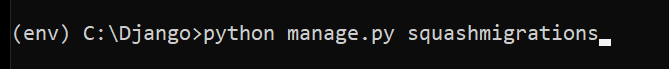
Если в модели неоднократно вносились изменения, после чего на их основе генерировались миграции, то таковых может накопиться довольно много. Чтобы уменьшить количество миграций и заодно ускорить их выполнение на ’’свежей” базе данных, рекомендуется осуществить слияние миграций— объединение их в одну.

Для этого достаточно отдать команду squashmigrations утилиты manage.py:

Manage.py squashmigrations <псевдоним приложения> [<имя первой миграции>]

<имя последней миграции> [—squashed\_name <имя результирующей миграции>]

[—no-optimize] [—noinput] [—no-input] [—no-header]

****

Обязательными ДЛЯ указания ЯВЛЯЮТСЯ ТОЛЬКО псевдоним приложения И имя последней миграции из числа подвергаемых слиянию. В этом случае будут обработаны все Миграции, начиная с самой первой из сформированных (обычно это начальная Миграция) и заканчивая указанной в команде. Пример:

Manage.py squashmigrations bboard 0004

Если задать имя первой миграции из подвергаемых слиянию, то будут обработаны Миграции, начиная с нее. Более ранние миграции будут пропущены. Пример:

Manage.py squashmigrations testapp 0002 0004

Рассмотрим ключи, поддерживаемые командой:

□ —squashed name — задает имя миграции, которая будет получена в результате слияния. Если оно отсутствует, то модуль результирующей миграции получит имя вида <имя первой миграции>\_вциавьед\_<имя последней миграции>.ру;

□ -no-optimize— отменяет оптимизацию кода миграции, что применяется для уменьшения его объема и повышения быстродействия. Рекомендуется указывать этот ключ, если слияние миграций выполнить не удалось или если результирующая миграция оказалась неработоспособной;

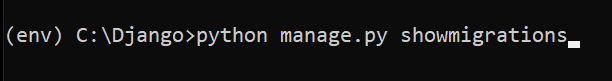
□ —noinput или -no-input — отключает вывод на экран сведений о слиянии миграций;

□ —no-header (начиная с Django 2.2) — предписывает не вставлять в начало модуля миграции, получаемой в результате слияния, комментарий с указанием версии Django и временной отметки слияния.

1. Вывод списка миграций

Чтобы просмотреть список всех миграций, имеющихся в проекте, следует отдать команду showmigrations утилиты manage.py:

Manage.py showmigrations [<список псевдонимов приложений, разделенных пробелами>] [—plan] [—р]



Если не указать псевдоним приложения, будут выведены все имеющиеся в проекте миграции с разбиением по приложениям. Если указать псевдоним приложения, то будут выведены только миграции из этого приложения. При задании списка псевдонимов приложений, разделенных пробелами, ВЫВОДЯТСЯ ТОЛЬКО миграции ИЗ ЭТИХ приложений, опять же, с разбиением по отдельным приложениям.

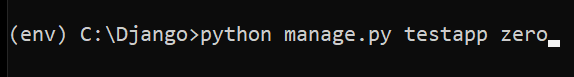
Список миграций при выводе сортируется в алфавитном порядке. Левее имени каждой миграции выводится значок [X], если миграция была выполнена, и [ ] —в противном случае.

Команда showmigrations поддерживает ключи —plan и -р, указывающие команде вместо списка вывести план миграций. План представляет собой список, отсортированный в последовательности, в которой Django будет выполнять миграции.

1. Отмена всех миграций

Наконец, Django позволяет нам отменить все миграции в приложении, тем самым удалив все созданные ими в базе данных структуры. Для этого достаточно выполнить команду migrate утилиты manage.py, указав в ней имя нужного приложения и zero в качестве имени миграции:

Manage.py migrate testapp zero



К сожалению, отменить отдельную, произвольно выбранную миграцию невозможно.