Django-filter is a generic, reusable application to alleviate writing some of the more mundane bits of view code. Specifically, it allows users to filter down a queryset based on a model’s fields, displaying the form to let them do this.
Django-filter can be installed from PyPI with tools like `pip`:

```
$ pip install django-filter
```

Then add `'django_filters'` to your `INSTALLED_APPS`.

```
INSTALLED_APPS = [
    ...
    'django_filters',
]
```

### 1.1 Requirements

Django-filter is tested against all supported versions of Python and Django, as well as the latest version of Django REST Framework (DRF).

- **Python**: 3.5, 3.6, 3.7, 3.8
- **Django**: 1.11, 2.0, 2.1, 2.2, 3.0
- **DRF**: 3.10+
Django-filter provides a simple way to filter down a queryset based on parameters a user provides. Say we have a `Product` model and we want to let our users filter which products they see on a list page.

**Note:** If you’re using django-filter with Django Rest Framework, it’s recommended that you read the integration docs after this guide.

### 2.1 The model

Let’s start with our model:

```python
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=255)
    price = models.DecimalField()
    description = models.TextField()
    release_date = models.DateField()
    manufacturer = models.ForeignKey(Manufacturer)
```

### 2.2 The filter

We have a number of fields and we want to let our users filter based on the name, the price or the `release_date`. We create a `FilterSet` for this:

```python
import django_filters

class ProductFilter(django_filters.FilterSet):
```

(continues on next page)
As you can see this uses a very similar API to Django’s ModelForm. Just like with a ModelForm we can also override filters, or add new ones using a declarative syntax.

### 2.2.1 Declaring filters

The declarative syntax provides you with the most flexibility when creating filters, however it is fairly verbose. We’ll use the below example to outline the core filter arguments on a FilterSet:

```python
class ProductFilter(django_filters.FilterSet):
    price = django_filters.NumberFilter()
    price__gt = django_filters.NumberFilter(field_name='price', lookup_expr='gt')
    price__lt = django_filters.NumberFilter(field_name='price', lookup_expr='lt')
    release_year = django_filters.NumberFilter(field_name='release_date', lookup_expr='year')
    release_year__gt = django_filters.NumberFilter(field_name='release_date', lookup_expr='year__gt')
    release_year__lt = django_filters.NumberFilter(field_name='release_date', lookup_expr='year__lt')
    manufacturer__name = django_filters.CharFilter(lookup_expr='icontains')

    class Meta:
        model = Product
```

There are two main arguments for filters:

- **field_name**: The name of the model field to filter on. You can traverse “relationship paths” using Django’s `__` syntax to filter fields on a related model. ex, `manufacturer__name`.
- **lookup_expr**: The field lookup to use when filtering. Django’s `__` syntax can again be used in order to support lookup transforms. ex, `year__gte`.

Together, the field `field_name` and `lookup_expr` represent a complete Django lookup expression. A detailed explanation of lookup expressions is provided in Django’s lookup reference. django-filter supports expressions containing both transforms and a final lookup.

### 2.2.2 Generating filters with Meta.fields

The FilterSet Meta class provides a `fields` attribute that can be used for easily specifying multiple filters without significant code duplication. The base syntax supports a list of multiple field names:

```python
import django_filters

class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = ['price', 'release_date']
```
The above generates ‘exact’ lookups for both the ‘price’ and ‘release_date’ fields.

Additionally, a dictionary can be used to specify multiple lookup expressions for each field:

```python
import django_filters
class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = {
            'price': ['lt', 'gt'],
            'release_date': ['exact', 'year__gt'],
        }
```

The above would generate ‘price__lt’, ‘price__gt’, ‘release_date’, and ‘release_date__year__gt’ filters.

**Note:** The filter lookup type ‘exact’ is an implicit default and therefore never added to a filter name. In the above example, the release date’s exact filter is ‘release_date’, not ‘release_date__exact’.

Items in the `fields` sequence in the `Meta` class may include “relationship paths” using Django’s `__` syntax to filter on fields on a related model:

```python
class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = ['manufacturer__country']
```

### Overriding default filters

Like `django.contrib.admin.ModelAdmin`, it is possible to override default filters for all the models fields of the same kind using `filter_overrides` on the `Meta` class:

```python
class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = {
            'name': ['exact'],
            'release_date': ['isnull'],
        }
        filter_overrides = {
            models.CharField: {
                'filter_class': django_filters.CharFilter,
                'extra': lambda f: {
                    'lookup_expr': 'icontains',
                },
            },
            models.BooleanField: {
                'filter_class': django_filters.BooleanFilter,
                'extra': lambda f: {
                    'widget': forms.CheckboxInput,
                },
            },
        }
```

---

2.2. The filter

7
2.2.3 Request-based filtering

The FilterSet may be initialized with an optional request argument. If a request object is passed, then you may access the request during filtering. This allows you to filter by properties on the request, such as the currently logged-in user or the Accepts-Languages header.

Note: It is not guaranteed that a request will be provided to the FilterSet instance. Any code depending on a request should handle the None case.

Filtering the primary .qs

To filter the primary queryset by the request object, simply override the FilterSet.qs property. For example, you could filter blog articles to only those that are published and those that are owned by the logged-in user (presumably the author’s draft articles).

```python
class ArticleFilter(django_filters.FilterSet):

    class Meta:
        model = Article
        fields = [...]

    @property
def qs(self):
        parent = super().qs
        author = getattr(self.request, 'user', None)

        return parent.filter(is_published=True) | parent.filter(author=author)
```

Filtering the related queryset for ModelChoiceFilter

The queryset argument for ModelChoiceFilter and ModelMultipleChoiceFilter supports callable behavior. If a callable is passed, it will be invoked with the request as its only argument. This allows you to perform the same kinds of request-based filtering without resorting to overriding FilterSet.__init__.

```python
def departments(request):
    if request is None:
        return Department.objects.none()

    company = request.user.company
    return company.department_set.all()

class EmployeeFilter(filters.FilterSet):
    department = filters.ModelChoiceFilter(queryset=departments)
    ...
```

2.2.4 Customize filtering with Filter.method

You can control the behavior of a filter by specifying a method to perform filtering. View more information in the method reference. Note that you may access the filterset’s properties, such as the request.
```python
class F(django_filters.FilterSet):
    username = CharFilter(method='my_custom_filter')

class Meta:
    model = User
    fields = ['username']

def my_custom_filter(self, queryset, name, value):
    return queryset.filter(**{
        name: value,
    })
```

### 2.3 The view

Now we need to write a view:

```python
def product_list(request):
    f = ProductFilter(request.GET, queryset=Product.objects.all())
    return render(request, 'my_app/template.html', {'filter': f})
```

If a queryset argument isn't provided then all the items in the default manager of the model will be used.

If you want to access the filtered objects in your views, for example if you want to paginate them, you can do that. They are in f.qs

### 2.4 The URL conf

We need a URL pattern to call the view:

```python
url(r'^list$', views.product_list)
```

### 2.5 The template

And lastly we need a template:

```html
{% extends "base.html" %}

{% block content %}
    <form action="" method="get">
        {{ filter.form.as_p }}
        <input type="submit" />
    </form>
    {% for obj in filter.qs %}
        {{ obj.name }} - ${{ obj.price }}<br />
    {% endfor %}
{% endblock %}
```

And that’s all there is to it! The form attribute contains a normal Django form, and when we iterate over the FilterSet.qs we get the objects in the resulting queryset.
2.6 Generic view & configuration

In addition to the above usage there is also a class-based generic view included in django-filter, which lives at `django_filters.views.FilterView`. You must provide either a `model` or `filterset_class` argument, similar to `ListView` in Django itself:

```python
# urls.py
from django.conf.urls import url
from django_filters.views import FilterView
from myapp.models import Product

urlpatterns = [
    url(r'^list/$', FilterView.as_view(model=Product)),
]
```

If you provide a `model` optionally you can set `filterset_fields` to specify a list or a tuple of the fields that you want to include for the automatic construction of the filterset class.

You must provide a template at `<app>/<model>_filter.html` which gets the context parameter `filter`. Additionally, the context will contain `object_list` which holds the filtered queryset.

A legacy functional generic view is still included in django-filter, although its use is deprecated. It can be found at `django_filters.views.object_filter`. You must provide the same arguments to it as the class based view:

```python
# urls.py
from django.conf.urls import url
from django_filters.views import object_filter
from myapp.models import Product

urlpatterns = [
    url(r'^list/$', object_filter, {'model': Product}),
]
```

The needed template and its context variables will also be the same as the class-based view above.
Integration with Django Rest Framework is provided through a DRF-specific FilterSet and a filter backend. These may be found in the rest_framework sub-package.

### 3.1 Quickstart

Using the new FilterSet simply requires changing the import path. Instead of importing from django_filters, import from the rest_framework sub-package.

```python
from django_filters import rest_framework as filters
class ProductFilter(filters.FilterSet):
    ...
```

Your view class will also need to add DjangoFilterBackend to the filter_backends.

```python
from django_filters import rest_framework as filters
class ProductList(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (filters.DjangoFilterBackend,)
    filterset_fields = ('category', 'in_stock')
```

If you want to use the django-filter backend by default, add it to the DEFAULT_FILTER_BACKENDS setting.

```python
# settings.py
INSTALLED_APPS = [
    ...
    'rest_framework',
    'django_filters',
]
```

(continues on next page)
3.2 Adding a FilterSet with `filterset_class`

To enable filtering with a FilterSet, add it to the `filterset_class` parameter on your view class.

```python
from rest_framework import generics
from django_filters import rest_framework as filters
from myapp import Product

class ProductFilter(filters.FilterSet):
    min_price = filters.NumberFilter(field_name="price", lookup_expr='gte')
    max_price = filters.NumberFilter(field_name="price", lookup_expr='lte')

    class Meta:
        model = Product
        fields = ['category', 'in_stock', 'min_price', 'max_price']

class ProductList(generics.ListAPIView):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
    filter_backends = (filters.DjangoFilterBackend,)
    filterset_class = ProductFilter
```

3.3 Using the `filterset_fields` shortcut

You may bypass creating a FilterSet by instead adding `filterset_fields` to your view class. This is equivalent to creating a FilterSet with just `Meta.fields`.

```python
from rest_framework import generics
from django_filters import rest_framework as filters
from myapp import Product

class ProductList(generics.ListAPIView):
    queryset = Product.objects.all()
    filter_backends = (filters.DjangoFilterBackend,)
    filterset_fields = ('category', 'in_stock')
    # Equivalent FilterSet:
    class ProductFilter(filters.FilterSet):
        class Meta:
            model = Product
            fields = ('category', 'in_stock')
```
Note that using `filterset_fields` and `filterset_class` together is not supported.

### 3.4 Overriding FilterSet creation

FilterSet creation can be customized by overriding the following methods on the backend class:

- `.get_filterset(self, request, queryset, view)`
- `.get_filterset_class(self, view, queryset=None)`
- `.get_filterset_kwargs(self, request, queryset, view)`

You can override these methods on a case-by-case basis for each view, creating unique backends, or these methods can be used to write your own hooks to the view class.

```python
class MyFilterBackend(filters.DjangoFilterBackend):
    def get_filterset_kwargs(self, request, queryset, view):
        kwargs = super().get_filterset_kwargs(request, queryset, view)

        # merge filterset kwargs provided by view class
        if hasattr(view, 'get_filterset_kwargs'):
            kwargs.update(view.get_filterset_kwargs())

        return kwargs
```

```python
class BooksFilter(filters.FilterSet):
    def __init__(self, *args, author=None, **kwargs):
        super().__init__(*args, **kwargs)

        # do something w/ author
```

```python
class BookViewSet(viewsets.ModelViewSet):
    filter_backends = [MyFilterBackend]
    filterset_class = BookFilter

    def get_filterset_kwargs(self):
        return {
            'author': self.get_author(),
        }
```

### 3.5 Schema Generation with Core API and Open API

The backend class integrates with DRF’s schema generation by implementing `get_schema_fields()` and `get_schema_operation_parameters()`. `get_schema_fields()` is automatically enabled when Core API is installed. `get_schema_operation_parameters()` is always enabled for Open API (new since DRF 3.9). Schema generation usually functions seamlessly, however the implementation does expect to invoke the view’s `get_queryset()` method. There is a caveat in that views are artificially constructed during schema generation, so the `args` and `kwargs` attributes will be empty. If you depend on arguments parsed from the URL, you will need to handle their absence in `get_queryset()`.

For example, your get queryset method may look like this:
class IssueViewSet(views.ModelViewSet):
    queryset = models.Issue.objects.all()

    def get_project(self):
        return models.Project.objects.get(pk=self.kwargs['project_id'])

    def get_queryset(self):
        project = self.get_project()

        return self.queryset \
            .filter(project=project) \
            .filter(author=self.request.user)

This could be rewritten like so:

class IssueViewSet(views.ModelViewSet):
    queryset = models.Issue.objects.all()

    def get_project(self):
        try:
            return models.Project.objects.get(pk=self.kwargs['project_id'])
        except models.Project.DoesNotExist:
            return None

    def get_queryset(self):
        project = self.get_project()

        if project is None:
            return self.queryset.none()

        return self.queryset \
            .filter(project=project) \
            .filter(author=self.request.user)

Or more simply as:

class IssueViewSet(views.ModelViewSet):
    queryset = models.Issue.objects.all()

    def get_queryset(self):
        if project_id is None:
            # project_id may be None
            return self.queryset \
                .filter(project_id=self.kwargs.get('project_id')) \
                .filter(author=self.request.user)

3.6 Crispy Forms

If you are using DRF’s browsable API or admin API you may also want to install django-crispy-forms, which will enhance the presentation of the filter forms in HTML views, by allowing them to render Bootstrap 3 HTML. Note that this isn’t actively supported, although pull requests for bug fixes are welcome.

pip install django-crispy-forms

With crispy forms installed and added to Django’s INSTALLED_APPS, the browsable API will present a filtering control for DjangoFilterBackend, like so:
3.7 Additional FilterSet Features

The following features are specific to the rest framework FilterSet:

- BooleanFilter's use the API-friendly BooleanWidget, which accepts lowercase true/false.
- Filter generation uses IsoDateTimeFilter for datetime model fields.
- Raised ValidationError's are reraised as their DRF equivalent.
4.1 Common problems for declared filters

Below are some of the common problem that occur when declaring filters. It is recommended that you read this as it provides a more complete understanding of how filters work.

4.1.1 Filter field_name and lookup_expr not configured

While field_name and lookup_expr are optional, it is recommended that you specify them. By default, if field_name is not specified, the filter’s name on the filterset class will be used. Additionally, lookup_expr defaults to exact. The following is an example of a misconfigured price filter:

```python
class ProductFilter(django_filters.FilterSet):
    price__gt = django_filters.NumberFilter()
```

The filter instance will have a field name of price__gt and an exact lookup type. Under the hood, this will incorrectly be resolved as:

```
Produce.objects.filter(price__gt__exact=value)
```

The above will most likely generate a FieldError. The correct configuration would be:

```python
class ProductFilter(django_filters.FilterSet):
    price__gt = django_filters.NumberFilter(field_name='price', lookup_expr='gt')
```

4.1.2 Missing lookup_expr for text search filters

It’s quite common to forget to set the lookup expression for CharField and TextField and wonder why a search for “foo” does not return results for “foobar”. This is because the default lookup type is exact, but you probably want to perform an icontains lookup.
4.1.3 Filter and lookup expression mismatch (in, range, isnull)

It’s not always appropriate to directly match a filter to its model field’s type, as some lookups expect different types of values. This is a commonly found issue with `in`, `range`, and `isnull` lookups. Let’s look at the following product model:

```python
class Product(models.Model):
    category = models.ForeignKey(Category, null=True)
```

Given that `category` is optional, it’s reasonable to want to enable a search for uncategorized products. The following is an incorrectly configured `isnull` filter:

```python
class ProductFilter(django_filters.FilterSet):
    uncategorized = django_filters.NumberFilter(field_name='category', lookup_expr='isnull')
```

So what’s the issue? While the underlying column type for `category` is an integer, `isnull` lookups expect a boolean value. A `NumberFilter` however only validates numbers. Filters are not ‘expression aware’ and won’t change behavior based on their `lookup_expr`. You should use filters that match the data type of the lookup expression instead of the data type underlying the model field. The following would correctly allow you to search for both uncategorized products and products for a set of categories:

```python
class NumberInFilter(django_filters.BaseInFilter, django_filters.NumberFilter):
    pass

class ProductFilter(django_filters.FilterSet):
    categories = NumberInFilter(field_name='category', lookup_expr='in')
    uncategorized = django_filters.BooleanFilter(field_name='category', lookup_expr='isnull')
```

More info on constructing `in` and `range` csv filters.

4.2 Filtering by empty values

There are a number of cases where you may need to filter by empty or null values. The following are some common solutions to these problems:

4.2.1 Filtering by null values

As explained in the above “Filter and lookup expression mismatch” section, a common problem is how to correctly filter by null values on a field.

Solution 1: Using a `BooleanFilter` with `isnull`

Using `BooleanFilter` with an `isnull` lookup is a builtin solution used by the FilterSet’s automatic filter generation. To do this manually, simply add:

```python
class ProductFilter(django_filters.FilterSet):
    uncategorized = django_filters.BooleanFilter(field_name='category', lookup_expr='isnull')
```
Note: Remember that the filter class is validating the input value. The underlying type of the mode field is not relevant here.

You may also reverse the logic with the exclude parameter.

```python
class ProductFilter(django_filters.FilterSet):
    has_category = django_filters.BooleanFilter(field_name='category', lookup_expr=__isnull__, exclude=True)
```

**Solution 2: Using ChoiceFilter’s null choice**

If you’re using a ChoiceFilter, you may also filter by null values by enabling the null_label parameter. More details in the ChoiceFilter reference docs.

```python
class ProductFilter(django_filters.FilterSet):
    category = django_filters.ModelChoiceFilter(
        field_name='category', lookup_expr='isnull',
        null_label='Uncategorized',
        queryset=Category.objects.all(),
    )
```

**Solution 3: Combining fields w/ MultiValueField**

An alternative approach is to use Django’s MultiValueField to manually add a BooleanField to handle null values. Proof of concept: https://github.com/carltongibson/django-filter/issues/446

### 4.2.2 Filtering by an empty string

It’s not currently possible to filter by an empty string, since empty values are interpreted as a skipped filter.

GET http://localhost/api/my-model?myfield=

**Solution 1: Magic values**

You can override the filter() method of a filter class to specifically check for magic values. This is similar to the ChoiceFilter’s null value handling.

GET http://localhost/api/my-model?myfield=EMPTY

```python
class MyCharFilter(filters.CharFilter):
    empty_value = 'EMPTY'

    def filter(self, qs, value):
        if value != self.empty_value:
            return super().filter(qs, value)
        qs = self.get_method(qs)(**{'%s__%s' % (self.field_name, self.lookup_expr): ''})
        return qs.distinct() if self.distinct else qs
```
Solution 2: Empty string filter

It would also be possible to create an empty value filter that exhibits the same behavior as an `isnull` filter.

```python
GET http://localhost/api/my-model?myfield__isempty=false
```

```python
from django.core.validators import EMPTY_VALUES

class EmptyStringFilter(filters.BooleanFilter):
    def filter(self, qs, value):
        if value in EMPTY_VALUES:
            return qs

        exclude = self.exclude ^ (value is False)
        method = qs.exclude if exclude else qs.filter

        return method(**{self.field_name: ''})

class MyFilterSet(filters.FilterSet):
    myfield__isempty = EmptyStringFilter(field_name='myfield')

class Meta:
    model = MyModel
```

4.3 Using initial values as defaults

In pre-1.0 versions of django-filter, a filter field’s initial value was used as a default when no value was submitted. This behavior was not officially supported and has since been removed.

**Warning:** It is recommended that you do **NOT** implement the below as it adversely affects usability. Django forms don’t provide this behavior for a reason.

- Using initial values as defaults is inconsistent with the behavior of Django forms.
- Default values prevent users from filtering by empty values.
- Default values prevent users from skipping that filter.

If defaults are necessary though, the following should mimic the pre-1.0 behavior:

```python
class BaseFilterSet(FilterSet):
    def __init__(self, data=None, *args, **kwargs):
        # if filterset is bound, use initial values as defaults
        if data is not None:
            # get a mutable copy of the QueryDict
            data = data.copy()

            for name, f in self.base_filters.items():
                initial = f.extra.get('initial')

                # filter param is either missing or empty, use initial as default
                if not data.get(name) and initial:
                    data[name] = initial
```

(continues on next page)
super().__init__(data, *args, **kwargs)
5.1 Enabling warnings

To view deprecations, you may need to enable warnings within python. This can be achieved with either the `-W` flag, or with `PYTHONWARNINGS` environment variable. For example, you could run your test suite like so:

```bash
$ python -W once manage.py test
```

The above would print all warnings once when they first occur. This is useful to know what violations exist in your code (or occasionally in third party code). However, it only prints the last line of the stack trace. You can use the following to raise the full exception instead:

```bash
$ python -W error manage.py test
```

5.2 Migrating to 2.0

This release contains several changes that break forwards compatibility. This includes removed features, renamed attributes and arguments, and some reworked features. Due to the nature of these changes, it is not feasible to release a fully forwards-compatible migration release. Please review the below list of changes and update your code accordingly.

5.2.1 `Filter.lookup_expr` list form removed (#851)

The `Filter.lookup_expr` argument no longer accepts `None` or a list of expressions. Use the `LookupChoiceFilter` instead.

5.2.2 `FilterSet filter_for_reverse_field` removed (#915)

The `filter_for_field` method now generates filters for reverse relationships, removing the need for `filter_for_reverse_field`. As a result, reverse relationships now also obey `Meta.filter_overrides`. 
5.2.3 View attributes renamed (#867)

Several view-related attributes have been renamed to improve consistency with other parts of the library. The following classes are affected:

- DRF ViewSet.filter_class => filterset_class
- DRF ViewSet.filter_fields => filterset_fields
- DjangoFilterBackend.default_filter_set => filterset_base
- DjangoFilterBackend.get_filter_class() => get_filterset_class()
- FilterMixin.filter_fields => filterset_fields

5.2.4 FilterSet Meta.together option removed (#791)

The Meta.together has been deprecated in favor of userland implementations that override the clean method of the Meta.form class. An example will be provided in a “recipes” section in future docs.

5.2.5 FilterSet “strictness” handling moved to view (#788)

Strictness handling has been removed from the FilterSet and added to the view layer. As a result, the FILTERS_STRICTNESS setting, Meta.strict option, and strict argument for the FilterSet initializer have all been removed.

To alter strictness behavior, the appropriate view code should be overridden. More details will be provided in future docs.

5.2.6 Filter.name renamed to Filter.field_name (#792)

The filter name has been renamed to field_name as a way to disambiguate the filter’s attribute name on its FilterSet class from the field_name used for filtering purposes.

5.2.7 Filter.widget and Filter.required removed (#734)

The filter class no longer directly stores arguments passed to its form field. All arguments are located in the filter's .extra dict.

5.2.8 MultiWidget replaced by SuffixedMultiWidget (#770)

RangeWidget, DateRangeWidget, and LookupTypeWidget now inherit from SuffixedMultiWidget, changing the suffixes of their query param names. For example, RangeWidget now has _min and _max suffixes instead of _0 and _1.

5.2.9 Filters like RangeFilter, DateRangeFilter, DateTimeFromToRangeFilter. .. (#770)

As they depend on MultiWidget, they need to be adjusted. In 1.0 release parameters were provided using _0 and _1 as suffix". For example, a parameter creation_date using “DateTimeFromToRangeFilter” will expect creation_date_after and creation_date_before instead of creation_date_0 and creation_date_1.
5.3 Migrating to 1.0

The 1.0 release of django-filter introduces several API changes and refinements that break forwards compatibility. Below is a list of deprecations and instructions on how to migrate to the 1.0 release. A forwards-compatible 0.15 release has also been created to help with migration. It is compatible with both the existing and new APIs and will raise warnings for deprecated behavior.

5.3.1 MethodFilter and Filter.action replaced by Filter.method (#382)

The functionality of MethodFilter and Filter.action has been merged together and replaced by the Filter.method parameter. The method parameter takes either a callable or the name of a FilterSet method. The signature now takes an additional name argument that is the name of the model field to be filtered on.

Since method is now a parameter of all filters, inputs are validated and cleaned by its field_class. The function will receive the cleaned value instead of the raw value.

```python
# 0.x
class UserFilter(FilterSet):
    last_login = filters.MethodFilter()

    def filter_last_login(self, qs, value):
        # try to convert value to datetime, which may fail.
        if value and looks_like_a_date(value):
            value = datetime(value)

        return qs.filter(last_login=value)

# 1.0
class UserFilter(FilterSet):
    last_login = filters.CharFilter(method='filter_last_login')

    def filter_last_login(self, qs, name, value):
        return qs.filter(**{name: value})
```

5.3.2 QuerySet methods are no longer proxied (#440)

The __iter__(), __len__(), __getitem__(), count() methods are no longer proxied from the queryset. To fix this, call the methods on the .qs property itself.

```python
f = UserFilter(request.GET, queryset=User.objects.all())

# 0.x
for obj in f:
    ...

# 1.0
for obj in f.qs:
    ...
```
5.3.3 Filters no longer autogenerated when Meta.fields is not specified (#450)

FilterSets had an undocumented behavior of autogenerating filters for all model fields when either `Meta.fields` was not specified or when set to `None`. This can lead to potentially unsafe data or schema exposure and has been deprecated in favor of explicitly setting `Meta.fields` to the `__all__` special value. You may also blacklist fields by setting the `Meta.exclude` attribute.

```python
class UserFilter(FilterSet):
    class Meta:
        model = User
        fields = '__all__'

# or
class UserFilter(FilterSet):
    class Meta:
        model = User
        exclude = ['password']
```

5.3.4 Move FilterSet options to Meta class (#430)

Several `FilterSet` options have been moved to the `Meta` class to prevent potential conflicts with declared filter names. This includes:

- `filter_overrides`
- `strict`
- `order_by_field`

```python
# 0.x
class UserFilter(FilterSet):
    filter_overrides = {}
    strict = STRICTNESS.RAISE_VALIDATION_ERROR
    order_by_field = 'order'
...

# 1.0
class UserFilter(FilterSet):
    ...
    class Meta:
        filter_overrides = {}
        strict = STRICTNESS.RAISE_VALIDATION_ERROR
        order_by_field = 'order'
```

5.3.5 FilterSet ordering replaced by OrderingFilter (#472)

The `FilterSet` ordering options and methods have been deprecated and replaced by `OrderingFilter`. Deprecated options include:

- `Meta.order_by`
- `Meta.order_by_field`

These options retain backwards compatibility with the following caveats:
• `order_by` asserts that `Meta.fields` is not using the dict syntax. This previously was undefined behavior, however the migration code is unable to support it.

• Prior, if no ordering was specified in the request, the FilterSet implicitly filtered by the first param in the `order_by` option. This behavior cannot be easily emulated but can be fixed by ensuring that the passed in queryset explicitly calls `order_by()`.

```python
filterset = MyFilterSet(queryset=MyModel.objects.order_by('field'))
```

The following methods are deprecated and will raise an assertion if present on the FilterSet:

• `.get_order_by()`
• `.get_ordering_field()`

To fix this, simply remove the methods from your class. You can subclass `OrderingFilter` to migrate any custom logic.

### 5.3.6 Deprecated `FILTERS_HELP_TEXT_FILTER` and `FILTERS_HELP_TEXT_EXCLUDE` (#437)

Generated filter labels in 1.0 will be more descriptive, including humanized text about the lookup being performed and if the filter is an exclusion filter.

These settings will no longer have an effect and will be removed in the 1.0 release.

### 5.3.7 DRF filter backend raises `TemplateDoesNotExist` exception (#562)

Templates are now provided by django-filter. If you are receiving this error, you may need to add `'django_filters'` to your `INSTALLED_APPS` setting. Alternatively, you could provide your own templates.
This document provides a guide on using additional FilterSet features.

6.1 Meta options

- **model**
- **fields**
- **exclude**
- **form**
- **filter_overrides**

6.1.1 Automatic filter generation with **model**

The FilterSet is capable of automatically generating filters for a given model’s fields. Similar to Django’s ModelForm, filters are created based on the underlying model field’s type. This option must be combined with either the **fields** or **exclude** option, which is the same requirement for Django’s ModelForm class, detailed here.

```python
class UserFilter(django_filters.FilterSet):
    class Meta:
        model = User
        fields = ['username', 'last_login']
```

6.1.2 Declaring filterable **fields**

The **fields** option is combined with **model** to automatically generate filters. Note that generated filters will not overwrite filters declared on the FilterSet. The **fields** option accepts two syntaxes:

- a list of field names
• a dictionary of field names mapped to a list of lookups

```python
class UserFilter(django_filters.FilterSet):
    class Meta:
        model = User
        fields = ['username', 'last_login']
```

# or

```python
class UserFilter(django_filters.FilterSet):
    class Meta:
        model = User
        fields = {
            'username': ['exact', 'contains'],
            'last_login': ['exact', 'year__gt'],
        }
```

The list syntax will create an exact lookup filter for each field included in `fields`. The dictionary syntax will create a filter for each lookup expression declared for its corresponding model field. These expressions may include both transforms and lookups, as detailed in the lookup reference.

### 6.1.3 Disable filter fields with `exclude`

The `exclude` option accepts a blacklist of field names to exclude from automatic filter generation. Note that this option will not disable filters declared directly on the `FilterSet`.

```python
class UserFilter(django_filters.FilterSet):
    class Meta:
        model = User
        exclude = ['password']
```

### 6.1.4 Custom Forms using `form`

The inner `Meta` class also takes an optional `form` argument. This is a form class from which `FilterSet.form` will subclass. This works similar to the `form` option on a `ModelAdmin`.

### 6.1.5 Customise filter generation with `filter_overrides`

The inner `Meta` class also takes an optional `filter_overrides` argument. This is a map of model fields to filter classes with options:

```python
class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = ['name', 'release_date']
        filter_overrides = {
            models.CharField: {
                'filter_class': django_filters.CharFilter,
                'extra': lambda f: {
                    'lookup_expr': 'icontains',
                },
            },
        }
```

(continues on next page)
6.2 Overriding FilterSet methods

When overriding classmethods, calling `super(MyFilterSet, cls)` may result in a `NameError` exception. This is due to the `FilterSetMeta` class calling these classmethods before the `FilterSet` class has been fully created. There are two recommended workarounds:

1. If using python 3.6 or newer, use the argumentless `super()` syntax.
2. For older versions of python, use an intermediate class. Ex:

```python
class Intermediate(django_filters.FilterSet):
    @classmethod
    def method(cls, arg):
        super(Intermediate, cls).method(arg)

    class Meta:
        model = Product
        fields = ['...']
```

6.2.1 filter_for_lookup()

Prior to version 0.13.0, filter generation did not take into account the `lookup_expr` used. This commonly caused malformed filters to be generated for `isnull`, `in`, and `range` lookups (as well as transformed lookups). The current implementation provides the following behavior:

- `isnull` lookups return a `BooleanFilter`
- `in` lookups return a filter derived from the CSV-based `BaseInFilter`.
- `range` lookups return a filter derived from the CSV-based `BaseRangeFilter`.

If you want to override the `filter_class` and `params` used to instantiate filters for a model field, you can override `filter_for_lookup()`. Ex:

```python
class ProductFilter(django_filters.FilterSet):
    class Meta:
        model = Product
        fields = {
            'release_date': ['exact', 'range'],
        }

    @classmethod
    def filter_for_lookup(cls, f, lookup_type):
        # ...
```

(continues on next page)
# override date range lookups
if isinstance(f, models.DateField) and lookup_type == 'range':
    return django_filters.DateRangeFilter, {}

# use default behavior otherwise
return super().filter_for_lookup(f, lookup_type)
This is a reference document with a list of the filters and their arguments.

### 7.1 Core Arguments

The following are the core arguments that apply to all filters. Note that they are joined to construct the complete lookup expression that is the left hand side of the ORM .filter() call.

#### 7.1.1 field_name

The name of the model field that is filtered against. If this argument is not provided, it defaults the filter’s attribute name on the FilterSet class. Field names can traverse relationships by joining the related parts with the ORM lookup separator (__.). e.g., a product’s manufacturer__.name.

#### 7.1.2 lookup_expr

The field lookup that should be performed in the filter call. Defaults to exact. The lookup_expr can contain transforms if the expression parts are joined by the ORM lookup separator (__.). e.g., filter a datetime by its year part year__.gt.

### 7.2 Keyword-only Arguments:

The following are optional arguments that can be used to modify the behavior of all filters.
7.2.1 label

The label as it will appear in the HTML, analogous to a form field's label argument. If a label is not provided, a
verbose label will be generated based on the field field_name and the parts of the lookup_expr (see: FILTERS_VERBOS_LOOKUPS).

7.2.2 method

An optional argument that tells the filter how to handle the queryset. It can accept either a callable or the name of a
method on the FilterSet. The callable receives a QuerySet, the name of the model field to filter on, and the
value to filter with. It should return a filtered Queryset.

Note that the value is validated by the Filter.field, so raw value transformation and empty value checking should
be unnecessary.

```python
class F(FilterSet):
    """Filter for Books by if books are published or not""
    published = BooleanFilter(field_name='published_on', method='filter_published')

    def filter_published(self, queryset, name, value):
        # construct the full lookup expression.
        lookup = '__'.join([name, 'isnull'])
        return queryset.filter(**{lookup: False})

        # alternatively, it may not be necessary to construct the lookup.
        return queryset.filter(published_on__isnull=False)

    class Meta:
        model = Book
        fields = ['published']

# Callables may also be defined out of the class scope.
def filter_not_empty(queryset, name, value):
    lookup = '__'.join([name, 'isnull'])
    return queryset.filter(**{lookup: False})

class F(FilterSet):
    """Filter for Books by if books are published or not""
    published = BooleanFilter(field_name='published_on', method=filter_not_empty)

    class Meta:
        model = Book
        fields = ['published']
```

7.2.3 distinct

A boolean that specifies whether the Filter will use distinct on the queryset. This option can be used to eliminate
duplicate results when using filters that span relationships. Defaults to False.

7.2.4 exclude

A boolean that specifies whether the Filter should use filter or exclude on the queryset. Defaults to False.
7.2.5 **kwargs

Any additional keyword arguments are stored as the `extra` parameter on the filter. They are provided to the accompanying form `Field` and can be used to provide arguments like `choices`. Some field-related arguments:

**widget**

The `django.form` Widget class which will represent the `Filter`. In addition to the widgets that are included with Django that you can use there are additional ones that `django-filter` provides which may be useful:

- **LinkWidget** – this displays the options in a manner similar to the way the Django Admin does, as a series of links. The link for the selected option will have `class="selected"`.
- **BooleanWidget** – this widget converts its input into Python’s True/False values. It will convert all case variations of `True` and `False` into the internal Python values.
- **CSVWidget** – this widget expects a comma separated value and converts it into a list of string values. It is expected that the field class handle a list of values as well as type conversion.
- **RangeWidget** – this widget is used with `RangeFilter` to generate two form input elements using a single field.

7.3 ModelChoiceFilter and ModelMultipleChoiceFilter arguments

These arguments apply specifically to `ModelChoiceFilter` and `ModelMultipleChoiceFilter` only.

7.3.1 queryset

`ModelChoiceFilter` and `ModelMultipleChoiceFilter` require a queryset to operate on which must be passed as a kwarg.

7.3.2 to_field_name

If you pass in `to_field_name` (which gets forwarded to the Django field), it will be used also in the default `get_filter_predicate` implementation as the model’s attribute.

7.4 Filters

7.4.1 CharFilter

This filter does simple character matches, used with `CharField` and `TextField` by default.

7.4.2 UUIDFilter

This filter matches UUID values, used with `models.UUIDField` by default.
7.4.3 BooleanFilter

This filter matches a boolean, either True or False, used with BooleanField and NullBooleanField by default.

7.4.4 ChoiceFilter

This filter matches values in its choices argument. The choices must be explicitly passed when the filter is declared on the FilterSet. For example,

```python
class User(models.Model):
    username = models.CharField(max_length=255)
    first_name = SubCharField(max_length=100)
    last_name = SubSubCharField(max_length=100)

    status = models.IntegerField(choices=STATUS_CHOICES, default=0)

STATUS_CHOICES = (
    (0, 'Regular'),
    (1, 'Manager'),
    (2, 'Admin'),
)

class F(FilterSet):
    status = ChoiceFilter(choices=STATUS_CHOICES)

class Meta:
    model = User
    fields = ['status']
```

ChoiceFilter also has arguments that enable a choice for not filtering, as well as a choice for filtering by None values. Each of the arguments have a corresponding global setting (Settings Reference).

- **empty_label**: The display label to use for the select choice to not filter. The choice may be disabled by setting this argument to None. Defaults to FILTERS_EMPTY_CHOICE_LABEL.

- **null_label**: The display label to use for the choice to filter by None values. The choice may be disabled by setting this argument to None. Defaults to FILTERS_NULL_CHOICE_LABEL.

- **null_value**: The special value to match to enable filtering by None values. This value defaults FILTERS_NULL_CHOICE_VALUE and needs to be a non-empty value ('', None, [], (), {}).

7.4.5 TypedChoiceFilter

The same as ChoiceFilter with the added possibility to convert value to match against. This could be done by using `coerce` parameter. An example use-case is limiting boolean choices to match against so only some predefined strings could be used as input of a boolean filter:

```python
import django_filters
from distutils.util import strtobool

BOOLEAN_CHOICES = ( ('false', 'False'), ('true', 'True'),)

class YourFilterSet(django_filters.FilterSet):
    ...
    flag = django_filters.TypedChoiceFilter(choices=BOOLEAN_CHOICES,
                                            coerce=strtobool)
```
7.4.6 MultipleChoiceFilter

The same as ChoiceFilter except the user can select multiple choices and the filter will form the OR of these choices by default to match items. The filter will form the AND of the selected choices when the conjoined=True argument is passed to this class.

Multiple choices are represented in the query string by reusing the same key with different values (e.g. ‘?status=Regular&status=Admin’).

distinct defaults to True as to-many relationships will generally require this.

Advanced Use: Depending on your application logic, when all or no choices are selected, filtering may be a noop. In this case you may wish to avoid the filtering overhead, particularly of the distinct call.

Set always_filter to False after instantiation to enable the default is_noop test.

Override is_noop if you require a different test for your application.

7.4.7 TypedMultipleChoiceFilter

Like MultipleChoiceFilter, but in addition accepts the coerce parameter, as in TypedChoiceFilter.

7.4.8 DateFilter

Matches on a date. Used with DateField by default.

7.4.9 TimeFilter

Matches on a time. Used with TimeField by default.

7.4.10 DateTimeFilter

Matches on a date and time. Used with DateTimeField by default.

7.4.11 IsoDateTimeFilter

Uses IsoDateTimeField to support filtering on ISO 8601 formatted dates, as are often used in APIs, and are employed by default by Django REST Framework.

Example:

class F(FilterSet):
    """Filter for Books by date published, using ISO 8601 formatted dates"""
    published = IsoDateTimeFilter()

    class Meta:
        model = Book
        fields = ['published']
7.4.12 DurationFilter

Matches on a duration. Used with `DurationField` by default.

Supports both Django ("%d %H:%M:%S.%f") and ISO 8601 formatted durations (but only the sections that are accepted by Python’s `timedelta`, so no year, month, and week designators, e.g. ‘P3DT10H22M’).

7.4.13 ModelChoiceFilter

Similar to a ChoiceFilter except it works with related models, used for `ForeignKey` by default.

If automatically instantiated, ModelChoiceFilter will use the default QuerySet for the related field. If manually instantiated you must provide the queryset kwarg.

Example:

```python
class F(FilterSet):
    """Filter for books by author""
    author = ModelChoiceFilter(queryset=Author.objects.all())

    class Meta:
        model = Book
        fields = ['author']
```

The queryset argument also supports callable behavior. If a callable is passed, it will be invoked with `FilterSet.request` as its only argument. This allows you to easily filter by properties on the request object without having to override the `FilterSet.__init__`.

**Note:** You should expect that the request object may be `None`.

```python
def departments(request):
    if request is None:
        return Department.objects.none()

    company = request.user.company
    return company.department_set.all()

class EmployeeFilter(filters.FilterSet):
    department = filters.ModelChoiceFilter(queryset=departments)
    ...
```

7.4.14 ModelMultipleChoiceFilter

Similar to a MultipleChoiceFilter except it works with related models, used for `ManyToManyField` by default.

As with ModelChoiceFilter, if automatically instantiated, ModelMultipleChoiceFilter will use the default QuerySet for the related field. If manually instantiated you must provide the queryset kwarg. Like ModelChoiceFilter, the queryset argument has callable behavior.

To use a custom field name for the lookup, you can use `to_field_name`:
```python
class FooFilter(BaseFilterSet):
    foo = django_filters.filters.ModelMultipleChoiceFilter(
        field_name='attr__uuid',
        to_field_name='uuid',
        queryset=Foo.objects.all(),
    )
```

If you want to use a custom queryset, e.g. to add annotated fields, this can be done as follows:

```python
class MyMultipleChoiceFilter(django_filters.ModelMultipleChoiceFilter):
    def get_filter_predicate(self, v):
        return {'annotated_field': v.annotated_field}

    def filter(self, qs, value):
        if value:
            qs = qs.annotate_with_custom_field()
            qs = super().filter(qs, value)
        return qs

foo = MyMultipleChoiceFilter(
    to_field_name='annotated_field',
    queryset=Model.objects.annotate_with_custom_field(),
)
```

The `annotate_with_custom_field` method would be defined through a custom QuerySet, which then gets used as the model’s manager:

```python
class CustomQuerySet(models.QuerySet):
    def annotate_with_custom_field(self):
        return self.annotate(
            custom_field=Case(
                When(foo_isnull=False, then=F('foo__uuid')),
                When(bar_isnull=False, then=F('bar__uuid')),
                default=None,
            ),
        )

class MyModel(models.Model):
    objects = CustomQuerySet.as_manager()
```

### 7.4.15 NumberFilter

Filters based on a numerical value, used with `IntegerField`, `FloatField`, and `DecimalField` by default.

### 7.4.16 NumericRangeFilter

Filters where a value is between two numerical values, or greater than a minimum or less than a maximum where only one limit value is provided. This filter is designed to work with the Postgres Numerical Range Fields, including `IntegerField`, `BigIntegerRangeField`, and `FloatRangeField` (available since Django 1.8). The default widget used is the `RangeField`.

Regular field lookups are available in addition to several containment lookups, including `overlap`, `contains`, and `contained_by`. More details in the Django docs.
If the lower limit value is provided, the filter automatically defaults to `startswith` as the lookup and `endswith` if only the upper limit value is provided.

### 7.4.17 RangeFilter

Filters where a value is between two numerical values, or greater than a minimum or less than a maximum where only one limit value is provided.

```python
class F(FilterSet):
    # "Filter for Books by Price"
    price = RangeFilter()

    class Meta:
        model = Book
        fields = ['price']

qs = Book.objects.all().order_by('title')

# Range: Books between 5€ and 15€
f = F({'price_min': '5', 'price_max': '15'}, queryset=qs)

# Min-Only: Books costing more the 11€
f = F({'price_min': '11'}, queryset=qs)

# Max-Only: Books costing less than 19€
f = F({'price_max': '19'}, queryset=qs)
```

### 7.4.18 DateRangeFilter

Filter similar to the admin changelist date one, it has a number of common selections for working with date fields.

### 7.4.19 DateFromToRangeFilter

Similar to a `RangeFilter` except it uses dates instead of numerical values. It can be used with `DateField`. It also works with `DateTimeField`, but takes into consideration only the date.

Example of using the `DateField` field:

```python
class Comment(models.Model):
    date = models.DateField()
    time = models.TimeField()

class F(FilterSet):
    date = DateFromToRangeFilter()

    class Meta:
        model = Comment
        fields = ['date']

# Range: Comments added between 2016-01-01 and 2016-02-01
f = F({'date_after': '2016-01-01', 'date_before': '2016-02-01'})

# Min-Only: Comments added after 2016-01-01
f = F({'date_after': '2016-01-01'})
```

(continues on next page)
# Max-Only: Comments added before 2016-02-01

\[ f = F({'date_before': '2016-02-01'}) \]

Note: When filtering ranges that occurs on DST transition dates `DateFromToRangeFilter` will use the first valid hour of the day for start datetime and the last valid hour of the day for end datetime. This is OK for most applications, but if you want to customize this behavior you must extend `DateFromToRangeFilter` and make a custom field for it.

Warning: If you’re using Django prior to 1.9 you may hit `AmbiguousTimeError` or `NonExistentTimeError` when start/end date matches DST start/end respectively. This occurs because versions before 1.9 don’t allow to change the DST behavior for making a datetime aware.

Example of using the `DateTimeField` field:

```python
class Article(models.Model):
    published = models.DateTimeField()

class F(FilterSet):
    published = DateFromToRangeFilter()

    class Meta:
        model = Article
        fields = ['published']

Article.objects.create(published='2016-01-01 8:00')
Article.objects.create(published='2016-01-20 10:00')
Article.objects.create(published='2016-02-10 12:00')

# Range: Articles published between 2016-01-01 and 2016-02-01
f = F({'published_after': '2016-01-01', 'published_before': '2016-02-01'})
assert len(f.qs) == 2

# Min-Only: Articles published after 2016-01-01
f = F({'published_after': '2016-01-01'})
assert len(f.qs) == 3

# Max-Only: Articles published before 2016-02-01
f = F({'published_before': '2016-02-01'})
assert len(f.qs) == 2
```

### 7.4.20 DateTimeFromToRangeFilter

Similar to a RangeFilter except it uses datetime format values instead of numerical values. It can be used with `DateTimeField`.

Example:

```python
class Article(models.Model):
    published = models.DateTimeField()
```

(continues on next page)
### 7.4.21 IsoDateTimeFromToRangeFilter

Similar to a `RangeFilter` except it uses ISO 8601 formatted values instead of numerical values. It can be used with `IsoDateTimeField`.

**Example:**

```python
class Article(models.Model):
    published = dajngo_filters.IsoDateTimeField()

class F(FilterSet):
    published = IsoDateTimeFromToRangeFilter()

    class Meta:
        model = Article
        fields = ['published']

Article.objects.create(published='2016-01-01T8:00:00+01:00')
Article.objects.create(published='2016-01-01T9:30:00+01:00')
Article.objects.create(published='2016-01-02T8:00:00+01:00')

# Range: Articles published 2016-01-01 between 8:00 and 10:00
f = F({'published_after': '2016-01-01T8:00:00+01:00', 'published_before': '2016-01-01T10:00:00+01:00'})
assert len(f.qs) == 2

# Min-Only: Articles published after 2016-01-01 8:00
f = F({'published_after': '2016-01-01T8:00:00+01:00'})
assert len(f.qs) == 3

# Max-Only: Articles published before 2016-01-01 10:00
f = F({'published_before': '2016-01-01T10:00:00+01:00'})
assert len(f.qs) == 2
```
7.4.22 TimeRangeFilter

Similar to a RangeFilter except it uses time format values instead of numerical values. It can be used with TimeField.

Example:

```python
class Comment(models.Model):
    date = models.DateField()
    time = models.TimeField()

class F(FilterSet):
    time = TimeRangeFilter()

class Meta:
    model = Comment
    fields = ['time']

# Range: Comments added between 8:00 and 10:00
f = F({'time_after': '8:00', 'time_before': '10:00'})

# Min-Only: Comments added after 8:00
f = F({'time_after': '8:00'})

# Max-Only: Comments added before 10:00
f = F({'time_before': '10:00'})
```

7.4.23 AllValuesFilter

This is a ChoiceFilter whose choices are the current values in the database. So if in the DB for the given field you have values of 5, 7, and 9 each of those is present as an option. This is similar to the default behavior of the admin.

7.4.24 AllValuesMultipleFilter

This is a MultipleChoiceFilter whose choices are the current values in the database. So if in the DB for the given field you have values of 5, 7, and 9 each of those is present as an option. This is similar to the default behavior of the admin.

7.4.25 LookupChoiceFilter

A combined filter that allows users to select the lookup expression from a dropdown.

- `lookup_choices` is an optional argument that accepts multiple input formats, and is ultimately normalized as the choices used in the lookup dropdown. See `.get_lookup_choices()` for more information.
- `field_class` is an optional argument that allows you to set the inner form field class used to validate the value. Default: `forms.CharField`

ex:

```python
price = django_filters.LookupChoiceFilter(
    field_class=forms.DecimalField,
    lookup_choices=[
        ('exact', 'Equals'),
    ],
)```

(continues on next page)
7.4.26 BaseInFilter

This is a base class used for creating IN lookup filters. It is expected that this filter class is used in conjunction with another filter class, as this class only validates that the incoming value is comma-separated. The secondary filter is then used to validate the individual values.

Example:

```python
class NumberInFilter(BaseInFilter, NumberFilter):
    pass

class F(FilterSet):
    id__in = NumberInFilter(field_name='id', lookup_expr='in')

    class Meta:
        model = User

User.objects.create(username='alex')
User.objects.create(username='jacob')
User.objects.create(username='aaron')
User.objects.create(username='carl')

# In: User with IDs 1 and 3.
f = F({'id__in': '1,3'})
assert len(f.qs) == 2
```

7.4.27 BaseRangeFilter

This is a base class used for creating RANGE lookup filters. It behaves identically to BaseInFilter with the exception that it expects only two comma-separated values.

Example:

```python
class NumberRangeFilter(BaseRangeFilter, NumberFilter):
    pass

class F(FilterSet):
    id__range = NumberRangeFilter(field_name='id', lookup_expr='range')

    class Meta:
        model = User

User.objects.create(username='alex')
User.objects.create(username='jacob')
User.objects.create(username='aaron')
User.objects.create(username='carl')

# Range: User with IDs between 1 and 3.
```
7.4.28 OrderingFilter

Enable queryset ordering. As an extension of ChoiceFilter it accepts two additional arguments that are used to build the ordering choices.

- **fields** is a mapping of {model field name: parameter name}. The parameter names are exposed in the choices and mask/alias the field names used in the order_by() call. Similar to field choices, fields accepts the 'list of two-tuples' syntax that retains order. fields may also just be an iterable of strings. In this case, the field names simply double as the exposed parameter names.

- **field_labels** is an optional argument that allows you to customize the display label for the corresponding parameter. It accepts a mapping of {field name: human readable label}. Keep in mind that the key is the field name, and not the exposed parameter name.

```python
class UserFilter(FilterSet):
    account = CharFilter(field_name='username')
    status = NumberFilter(field_name='status')

    o = OrderingFilter(
        fields=(
            ('username', 'account'),
            ('first_name', 'first_name'),
            ('last_name', 'last_name'),
        ),
        field_labels={
            'username': 'User account',
        }
    )

class Meta:
    model = User
    fields = ['first_name', 'last_name']

>>> UserFilter().filters['o'].field.choices
[['account', 'User account'],
['account', 'User account (descending)'],
['first_name', 'First name'],
['first_name', 'First name (descending)'],
['last_name', 'Last name'],
['last_name', 'Last name (descending)'],
]
```

Additionally, you can just provide your own choices if you require explicit control over the exposed options. For example, when you might want to disable descending sort options.

```python
class UserFilter(FilterSet):
    account = CharFilter(field_name='username')
    status = NumberFilter(field_name='status')
```

(continues on next page)
This filter is also CSV-based, and accepts multiple ordering params. The default select widget does not enable the use of this, but it is useful for APIs. SelectMultiple widgets are not compatible, given that they are not able to retain selection order.

**Adding Custom filter choices**

If you wish to sort by non-model fields, you’ll need to add custom handling to an `OrderingFilter` subclass. For example, if you want to sort by a computed ‘relevance’ factor, you would need to do something like the following:

```python
class CustomOrderingFilter(django_filters.OrderingFilter):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.extra['choices'] += [
            ('relevance', 'Relevance'),
            ('-relevance', 'Relevance (descending)'),
        ]

    def filter(self, qs, value):
        # OrderingFilter is CSV-based, so 'value' is a list
        if any(v in ['relevance', '-relevance'] for v in value):
            # sort queryset by relevance
            return ...

        return super().filter(qs, value)
```

```
8.1 IsoDateTimeField

Extends django.forms.DateTimeField to allow parsing ISO 8601 formatted dates, in addition to existing formats.

Defines a class level attribute ISO_8601 as constant for the format.

Sets input Formats = [ISO_8601] — this means that by default IsoDateTimeField will only parse ISO 8601 formatted dates.

You may set input Formats to your list of required formats as per the DateTimeField Docs, using the ISO_8601 class level attribute to specify the ISO 8601 format.

```python
f = IsoDateTimeField()
f.input_formats = [IsoDateTimeField.ISO_8601] + DateTimeField.input_formats
```
This is a reference document with a list of the provided widgets and their arguments.

9.1 LinkWidget

This widget renders each option as a link, instead of an actual <input>. It has one method that you can override for additional customizability. `option_string()` should return a string with 3 Python keyword argument placeholders:

1. `attrs`: This is a string with all the attributes that will be on the final <a> tag.
2. `query_string`: This is the query string for use in the `href` option on the <a> element.
3. `label`: This is the text to be displayed to the user.

9.2 BooleanWidget

This widget converts its input into Python’s True/False values. It will convert all case variations of True and False into the internal Python values. To use it, pass this into the `widgets` argument of the BooleanFilter:

```python
active = BooleanFilter(widget=BooleanWidget())
```

9.3 CSVWidget

This widget expects a comma separated value and converts it into a list of string values. It is expected that the field class handle a list of values as well as type conversion.
### 9.4 RangeWidget

This widget is used with RangeFilter and its subclasses. It generates two form input elements which generally act as start/end values in a range. Under the hood, it is Django's forms.TextInput widget and excepts the same arguments and values. To use it, pass it to `widget` argument of a RangeField:

```python
date_range = DateFromToRangeFilter(widget=RangeWidget(attrs={'placeholder': 'YYYY/MM/DD'}))
```

### 9.5 SuffixedMultiWidget

Extends Django's builtin MultiWidget to append custom suffixes instead of indices. For example, take a range widget that accepts minimum and maximum bounds. By default, the resulting query params would look like the following:

```
GET /products?price_0=10&price_1=25 HTTP/1.1
```

By using SuffixedMultiWidget instead, you can provide human-friendly suffixes.

```python
class RangeWidget(SuffixedMultiWidget):
    suffixes = ['min', 'max']
```

The query names are now a little more ergonomic.

```
GET /products?price_min=10&price_max=25 HTTP/1.1
```
Here is a list of all available settings of django-filters and their default values. All settings are prefixed with FILTERS_, although this is a bit verbose it helps to make it easy to identify these settings.

10.1 FILTERS_EMPTY_CHOICE_LABEL

Default: ‘---------’
Set the default value for ChoiceFilter.empty_label. You may disable the empty choice by setting this to None.

10.2 FILTERS_NULL_CHOICE_LABEL

Default: None
Set the default value for ChoiceFilter.null_label. You may enable the null choice by setting a non-None value.

10.3 FILTERS_NULL_CHOICE_VALUE

Default: 'null'
Set the default value for ChoiceFilter.null_value. You may want to change this value if the default 'null' string conflicts with an actual choice.

10.4 FILTERS_DISABLE_HELP_TEXT

Default: False
Some filters provide informational help_text. For example, csv-based filters (filters.BaseCSVFilter) inform users that “Multiple values may be separated by commas”.
You may set this to True to disable the help_text for all filters, removing the text from the rendered form’s output.

10.5 FILTERS_VERBOSE_LOOKUPS

Note: This is considered an advanced setting and is subject to change.

Default:

```python
# refer to 'django_filters.conf.DEFAULTS'
'VERBOSE_LOOKUPS': {
    'exact': _(''),
    'iexact': _(''),
    'contains': _('contains'),
    'icontains': _('contains'),
    ...
}
```

This setting controls the verbose output for generated filter labels. Instead of getting expression parts such as “lt” and “contained_by”, the verbose label would contain “is less than” and “is contained by”. Verbose output may be disabled by setting this to a falsy value.

This setting also accepts callables. The callable should not require arguments and should return a dictionary. This is useful for extending or overriding the default terms without having to copy the entire set of terms to your settings. For example, you could add verbose output for “exact” lookups.

```python
# settings.py
def FILTERS_VERBOSE_LOOKUPS():
    from django_filters.conf import DEFAULTS

    verbose_lookups = DEFAULTS['VERBOSE_LOOKUPS'].copy()
    verbose_lookups.update({
        'exact': 'is equal to',
    })
    return verbose_lookups
```
The easiest way to run the django-filter tests is to check out the source code and create a virtualenv where you can install the test dependencies. Django-filter uses a custom test runner to configure the environment, so a wrapper script is available to set up and run the test suite.

Note: The following assumes you have virtualenv and git installed.

### 11.1 Clone the repository

Get the source code using the following command:

```
$ git clone https://github.com/carltongibson/django-filter.git
```

Switch to the django-filter directory:

```
$ cd django-filter
```

### 11.2 Set up the virtualenv

Create a new virtualenv to run the test suite in:

```
$ virtualenv venv
```

Then activate the virtualenv and install the test requirements:

```
$ source venv/bin/activate
$ pip install -r requirements/test.txt
```
11.3 Execute the test runner

Run the tests with the runner script:

```
$ python runtests.py
```

11.4 Test all supported versions

You can also use the excellent tox testing tool to run the tests against all supported versions of Python and Django. Install tox, and then simply run:

```
$ pip install tox
$ tox
```

11.5 Housekeeping

The isort utility is used to maintain module imports. You can either test the module imports with the appropriate tox env, or with isort directly.

```
$ pip install tox
$ tox -e isort
  
# or

$ pip install isort
$ isort --check-only --recursive django_filters tests
```

To sort the imports, simply remove the --check-only option.

```
$ isort --recursive django_filters tests
```