django-tables2

Release 2.3.1

unknown

Sep 14, 2020

GETTING STARTED

Its features include:

- Any iterable can be a data-source, but special support for Django QuerySets is included.
- The built in UI does not rely on JavaScript.
- Support for automatic table generation based on a Django model.
- Supports custom column functionality via subclassing.
- Pagination.
- Column based table sorting.
- Template tag to enable trivial rendering to HTML.
- Generic view mixin.

About the app:

- Available on pypi
- Tested against currently supported versions of Django and the python versions Django supports
- Documentation on readthedocs.org
- Bug tracker

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Installation

Django-tables2 is Available on pypi and can be installed using pip:

```
pip install django-tables2
```

After installing, add `'django_tables2'` to `INSTALLED_APPS` and make sure that `"django.template.context_processors.request"` is added to the `context_processors` in your template setting `OPTIONS`.

1.2 Tutorial

This is a step-by-step guide to learn how to install and use django-tables2 using Django 2.0 or later.

1. `pip install django-tables2`
2. Start a new Django app using `python manage.py startapp tutorial`
3. Add both `"django_tables2"` and `"tutorial"` to your `INSTALLED_APPS` setting in `settings.py`.

Now, add a model to your `tutorial/models.py`:

```
# tutorial/models.py
class Person(models.Model):
    name = models.CharField(max_length=100, verbose_name="full name")
```

Create the database tables for the newly added model:

```
$ python manage.py makemigrations tutorial
$ python manage.py migrate tutorial
```

Add some data so you have something to display in the table:

```
$ python manage.py shell
>>> from tutorial.models import Person
>>> Person.objects.bulk_create([Person(name="Jieter"), Person(name="Bradley")])
[<Person: Person object>, <Person: Person object>]
```

Now use a generic `ListView` to pass a `Person` QuerySet into a template. Note that the context name used by `ListView` is `object_list` by default:

```
# tutorial/views.py
from django.views.generic import ListView
from .models import Person

class PersonListView(ListView):
    model = Person
    template_name = 'tutorial/people.html'
```

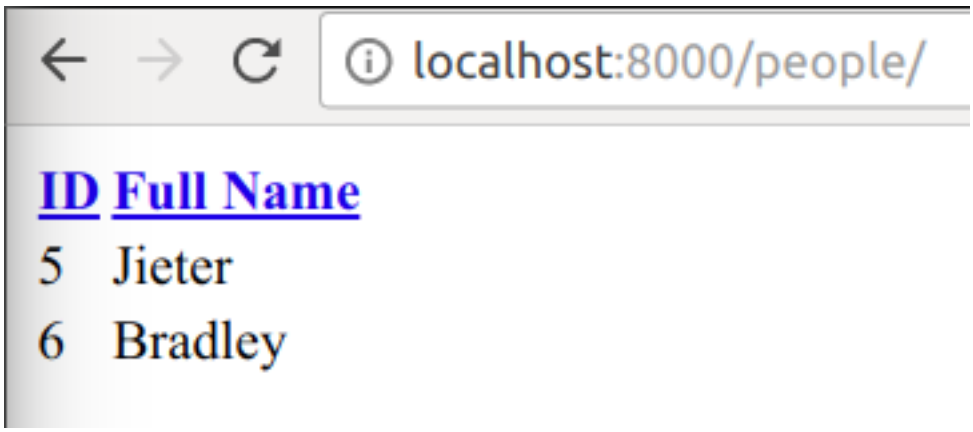
Add the view to your `urls.py`:

```
# urls.py
from django.urls import path
from django.contrib import admin

from tutorial.views import PersonListView

urlpatterns = [
    path("admin/", admin.site.urls),
    path("people/", PersonListView.as_view())
]
```

Finally, create the template:

```
{# tutorial/templates/tutorial/people.html #}
{% load render_table from django_tables2 %}
<!doctype html>
<html>
    <head>
        <title>List of persons</title>
    </head>
    <body>
        {% render_table object_list %}
    </body>
</html>
```

You should be able to load the page in the browser (http://localhost:8000/people/ by default), you should see:



This view supports pagination and ordering by default.

While simple, passing a QuerySet directly to `{% render_table %}` does not allow for any customization. For that, you must define a custom `Table` class:

```
# tutorial/tables.py
import django_tables2 as tables
from .models import Person

class PersonTable(tables.Table):
    class Meta:
        model = Person
        template_name = "django_tables2/bootstrap.html"
        fields = ("name", )
```

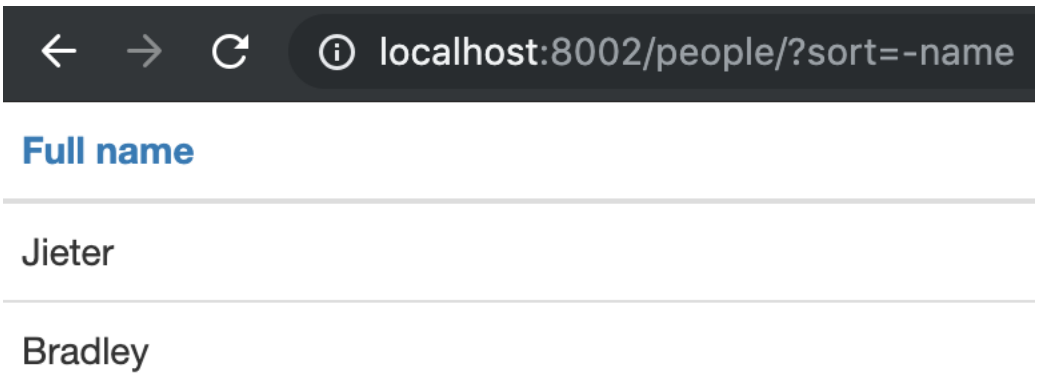
You will then need to instantiate and configure the table in the view, before adding it to the context:

```
# tutorial/views.py
from django_tables2 import SingleTableView

from .models import Person
from .tables import PersonTable


class PersonListView(SingleTableView):
    model = Person
    table_class = PersonTable
    template_name = 'tutorial/people.html'
```

Rather than passing a QuerySet to `{% render_table %}`, instead pass the table instance:

```
{# tutorial/templates/tutorial/people.html #}
{% load render_table from django_tables2 %}
<!doctype html>
<html>
    <head>
        <title>List of persons</title>
        <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/
↪css/bootstrap.min.css" />
    </head>
    <body>
        {% render_table table %}
    </body>
</html>
```

This results in a table rendered with the bootstrap3 style sheet:



At this point you have only changed the columns rendered in the table and the template. There are several topic you can read into to further customize the table:

- **Table data**
  - *Populating the table with data*,
  - *Filtering table data*
- **Customizing the rendered table**
  - *Headers and footers*
  - *Pinned rows*
- *API Reference*

If you think you don't have a lot customization to do and don't want to make a full class declaration use `django_tables2.tables.table_factory`.

1.3 Populating a table with data

Tables can be created from a range of input data structures. If you have seen the tutorial you will have seen a `QuerySet` being used, however any iterable that supports `len()` and contains items that exposes key-based access to column values is fine.

1.3.1 List of dicts

In an example we will demonstrate using list of dicts. When defining a table it is necessary to declare each column:

```
import django_tables2 as tables

data = [
    {"name": "Bradley"},
    {"name": "Stevie"},
]

class NameTable(tables.Table):
    name = tables.Column()

table = NameTable(data)
```

1.3.2 QuerySets

If you build use tables to display `QuerySet` data, rather than defining each column manually in the table, the `Table.Meta.model` option allows tables to be dynamically created based on a model:

```
# models.py
from django.contrib.auth import get_user_model
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    user = models.ForeignKey(get_user_model(), null=True, on_delete=models.CASCADE)
```

(continues on next page)

(continued from previous page)

```
    birth_date = models.DateField()

# tables.py
import django_tables2 as tables

class PersonTable(tables.Table):
    class Meta:
        model = Person

# views.py
def person_list(request):
    table = PersonTable(Person.objects.all())

    return render(request, "person_list.html", {
        "table": table
    })
```

This has a number of benefits:

- Less repetition
- Column headers are defined using the field's `verbose_name`
- Specialized columns are used where possible (e.g. *`DateColumn`* for a `DateField`)

When using this approach, the following options might be useful to customize what fields to show or hide:

- `sequence` – reorder columns (if used alone, columns that are not specified are still rendered in the table after the specified columns)
- `fields` – specify model fields to *include*
- `exclude` – specify model fields to *exclude*

These options can be specified as tuples. In this example we will demonstrate how this can be done:

```
# tables.py
class PersonTable(tables.Table):
    class Meta:
        model = Person
        sequence = ("last_name", "first_name", "birth_date", )
        exclude = ("user", )
```

With these options specified, the columns would be show according to the order defined in the `sequence`, while the `user` column will be hidden.

1.3.3 Performance

Django-tables tries to be efficient in displaying big datasets. It tries to avoid converting the `QuerySet` instances to lists by using SQL to slice the data and should be able to handle datasets with 100k records without a problem.

However, when performance is degrading, these tips might help:

1. For large datasets, try to use *`LazyPaginator`*.
2. Try to strip the table of customizations and check if performance improves. If so, re-add them one by one, checking for performance after each step. This should help to narrow down the source of your performance problems.

1.4 Alternative column data

Various options are available for changing the way the table is *rendered*. Each approach has a different balance of ease-of-use and flexibility.

1.4.1 Using `Accessors`

Each column has a 'key' that describes which value to pull from each record to populate the column's cells. By default, this key is just the name given to the column, but it can be changed to allow foreign key traversal or other complex cases.

To reduce ambiguity, rather than calling it a 'key', we use the name 'accessor'.

Accessors are just double-underscore separated paths that describe how an object should be traversed to reach a specific value, for example:

```
>>> from django_tables2 import A
>>> data = {"abc": {"one": {"two": "three"}}}
>>> A("abc__one__two").resolve(data)
'three'
```

The separators __ represent relationships, and are attempted in this order:

1. Dictionary lookup `a[b]`
2. Attribute lookup `a.b`
3. List index lookup `a[int(b)]`

If the resulting value is callable, it is called and the return value is used.

1.4.2 `Table.render_foo` methods

To change how a column is rendered, define a `render_foo` method on the table for example: `render_row_number()` for a column named `row_number`. This approach is suitable if you have a one-off change that you do not want to use in multiple tables or if you want to combine the data from two columns into one.

Supported keyword arguments include:

- `record` – the entire record for the row from the *table data*
- `value` – the value for the cell retrieved from the *table data*
- `column` – the *`Column`* object
- `bound_column` – the *`BoundColumn`* object
- `bound_row` – the *`BoundRow`* object
- `table` – alias for `self`

This example shows how to render the row number in the first row:

```
>>> import django_tables2 as tables
>>> import itertools
>>>
>>> class SimpleTable(tables.Table):
...     row_number = tables.Column(empty_values=())
...     id = tables.Column()
```

(continues on next page)

(continued from previous page)

```
...     age = tables.Column()
...
...     def __init__(self, *args, **kwargs):
...         super().__init__(*args, **kwargs)
...         self.counter = itertools.count()
...
...     def render_row_number(self):
...         return "Row %d" % next(self.counter)
...
...     def render_id(self, value):
...         return "<%s>" % value
...
>>> table = SimpleTable([{"age": 31, "id": 10}, {"age": 34, "id": 11}])
>>> print(", ".join(map(str, table.rows[0])))
Row 0, <10>, 31
```

Python's `inspect.getargspec` is used to only pass the arguments declared by the function. This means it's not necessary to add a catch all (`**`) keyword argument.

The `render_foo` method can also be used to combine data from two columns into one column. The following example shows how the the value for the `last_name` field is appended to the `name` field using the `render_name` function. Note that `value` is the value in the column and `record` is used to access the values in the `last_name` column:

```
# models.py
class Customers(models.Model):
    name = models.CharField(max_length=50, null=False, blank=False)
    last_name = models.CharField(max_length=50, null=False, blank=False)
    description = models.TextField(blank=True)

# tables.py
from .models import Customers
from django.utils.html import format_html

class CustomerTable(tables.Table):
    name = tables.Column()
    description = tables.Column()

    def render_name(self, value, record):
        return format_html("<b>{} {}</b>", value, record.last_name)
```

If you need to access logged-in user (or request in general) in your render methods, you can reach it through `self.request`:

```
def render_count(self, value):
    if self.request.user.is_authenticated():
        return value
    else:
        return '---'
```

Important: `render_foo` methods are *only* called if the value for a cell is determined to be not an *empty value*. When a value is in `Column.empty_values`, a default value is rendered instead (both *`Column.render`* and `Table.render_FOO` are skipped).

Important: `render_foo` methods determine what value is rendered, but which make sorting the column have unexpected results. In those cases, you might want to also define a *table.order_FOO() methods* method.

1.4.3 `Table.value_foo` methods

If you want to use `Table.as_values` to export your data, you might want to define a method `value_foo`, which is analogous to `render_foo`, but used to render the values rather than the HTML output.

Please refer to `Table.as_values` for an example.

1.4.4 Subclassing `Column`

Defining a column subclass allows functionality to be reused across tables. Columns have a `render` method that behaves the same as *Table.render_foo methods* methods on tables:

```
>>> import django_tables2 as tables
>>>
>>> class UpperColumn(tables.Column):
...     def render(self, value):
...         return value.upper()
...
>>> class Example(tables.Table):
...     normal = tables.Column()
...     upper = UpperColumn()
...
>>> data = [{"normal": "Hi there!",
...          "upper":  "Hi there!"}]
...
>>> table = Example(data)
>>> # renders to something like this:
'''<table>
    <thead><tr><th>Normal</th><th>Upper</th></tr></thead>
    <tbody><tr><td>Hi there!</td><td>HI THERE!</td></tr></tbody>
</table>'''
```

See *Table.render_foo methods* for a list of arguments that can be accepted.

For complicated columns, you may want to return HTML from the `render()` method. Make sure to use Django's html formatting functions:

```
>>> from django.utils.html import format_html
>>>
>>> class ImageColumn(tables.Column):
...     def render(self, value):
...         return format_html('<img src="/media/img/{}.jpg" />', value)
...
```

1.5 Alternative column ordering

When using QuerySet data, one might want to show a computed value which is not in the database. In this case, attempting to order the column will cause an exception:

```
# models.py
class Person(models.Model):
    first_name = models.CharField(max_length=200)
    family_name = models.CharField(max_length=200)

    @property
    def name(self):
        return "{} {}".format(self.first_name, self.family_name)

# tables.py
class PersonTable(tables.Table):
    name = tables.Column()
```

```
>>> table = PersonTable(Person.objects.all())
>>> table.order_by = "name"
>>>
>>> # will result in:
FieldError: Cannot resolve keyword 'name' into field. Choices are: first_name, family_
↪name
```

To prevent this, django-tables2 allows two ways to specify custom ordering: accessors and `order_FOO()` methods.

1.5.1 Ordering by accessors

You can supply an `order_by` argument containing a name or a tuple of the names of the columns the database should use to sort it:

```
class PersonTable(tables.Table):
    name = tables.Column(order_by=("first_name", "family_name"))
```

`Accessor` syntax can be used as well, as long as they point to a model field.

If ordering does not make sense for a particular column, it can be disabled via the `orderable` argument:

```
class SimpleTable(tables.Table):
    name = tables.Column()
    actions = tables.Column(orderable=False)
```

1.5.2 `table.order_FOO()` methods

Another solution for alternative ordering is being able to chain functions on to the original QuerySet. This method allows more complex functionality giving the ability to use all of Django's QuerySet API.

Adding a `Table.order_FOO` method (where `FOO` is the name of the column), gives you the ability to chain to, or modify, the original QuerySet when that column is selected to be ordered.

The method takes two arguments: `QuerySet`, and `is_descending`. The return must be a tuple of two elements. The first being the QuerySet and the second being a boolean; note that modified QuerySet will only be used if the boolean is `True`.

For example, let's say instead of ordering alphabetically, ordering by amount of characters in the first_name is desired. The implementation would look like this:

```
# tables.py
from django.db.models.functions import Length

class PersonTable(tables.Table):
    name = tables.Column()

    def order_name(self, queryset, is_descending):
        queryset = queryset.annotate(
            length=Length("first_name")
        ).order_by(("-" if is_descending else "") + "length")
        return (queryset, True)
```

As another example, presume the situation calls for being able to order by a mathematical expression. In this scenario, the table needs to be able to be ordered by the sum of both the shirts and the pants. The custom column will have its value rendered using *Table.render_foo methods*.

This can be achieved like this:

```
# models.py
class Person(models.Model):
    first_name = models.CharField(max_length=200)
    family_name = models.CharField(max_length=200)
    shirts = models.IntegerField()
    pants = models.IntegerField()


# tables.py
from django.db.models import F

class PersonTable(tables.Table):
    clothing = tables.Column()

    class Meta:
        model = Person

    def render_clothing(self, record):
        return str(record.shirts + record.pants)

    def order_clothing(self, queryset, is_descending):
        queryset = queryset.annotate(
            amount=F("shirts") + F("pants")
        ).order_by(("-" if is_descending else "") + "amount")
        return (queryset, True)
```

1.5.3 Using `Column.order()` on custom columns

If you created a custom column, which also requires custom ordering like explained above, you can add the body of your `order_foo` method to the order method on your custom column, to allow easier reuse.

For example, the `PersonTable` from above could also be defined like this:

```
class ClothingColumn(tables.Column):
    def render(self, record):
        return str(record.shirts + record.pants)

    def order(self, queryset, is_descending):
```

(continues on next page)

(continued from previous page)

```
        queryset = queryset.annotate(
            amount=F("shirts") + F("pants")
        ).order_by(("-" if is_descending else "") + "amount")
        return (queryset, True)


class PersonTable(tables.Table):
    clothing = ClothingColumn()

    class Meta:
        model = Person
```

1.6 Column and row attributes

1.6.1 Column attributes

Column attributes can be specified using the `dict` with specific keys. The dict defines HTML attributes for one of more elements within the column. Depending on the column, different elements are supported, however `th`, `td`, and `cell` are supported universally:

```
>>> import django_tables2 as tables
>>>
>>> class SimpleTable(tables.Table):
...     name = tables.Column(attrs={"th": {"id": "foo"}})
...
>>> # will render something like this:
'{snip}<thead><tr><th id="foo">{snip}<tbody><tr><td>{snip}'
```

Have a look at each column's API reference to find which elements are supported.

If you need to add some extra attributes to column's tags rendered in the footer, use key name `tf`, as described in section on *CSS*.

Callables passed in this dict will be called, with optional kwargs `table`, `bound_column` `record` and `value`, with the return value added. For example:

```
class Table(tables.Table):
    person = tables.Column(attrs={
        "td": {
            "data-length": lambda value: len(value)
        }
    })
```

will render the `<td>`'s in the tables `<body>` with a `data-length` attribute containing the number of characters in the value.

Note: The keyword arguments `record` and `value` only make sense in the context of a row containing data. If you supply a callable with one of these keyword arguments, it will not be executed for the header and footer rows.

If you also want to customize the attributes of those tags, you must define a callable with a catchall (`**kwargs`) argument:

```
def data_first_name(**kwargs):
    first_name = kwargs.get("value", None)
    if first_name is None:
        return "header"
    else:
        return first_name

class Table(tables.Table):
    first_name = tables.Column(attrs={
        "td": {
            'data-first-name': data_first_name
        }
    })
```

This `attrs` can also be defined when subclassing a column, to allow better reuse:

```
class PersonColumn(tables.Column):
    attrs = {
        "td": {
            "data-first-name": lambda record: record.first_name
            "data-last-name": lambda record: record.last_name
        }
    }
    def render(self, record):
        return "{} {}".format(record.first_name, record.last_name)

class Table(tables.Table):
    person = PersonColumn()
```

is equivalent to the previous example.

1.6.2 Row attributes

Row attributes can be specified using a dict defining the HTML attributes for the `<tr>` element on each row.

By default, class names *odd* and *even* are supplied to the rows, which can be customized using the `row_attrs` *`Table.Meta`* attribute or as argument to the constructor of `Table`. String-like values will just be added, callables will be called with optional keyword arguments `record` and `table`, the return value will be added. For example:

```
class Table(tables.Table):
    class Meta:
        model = User
        row_attrs = {
            "data-id": lambda record: record.pk
        }
```

will render tables with the following `<tr>` tag

```
<tr class="odd" data-id="1"> [...] </tr>
<tr class="even" data-id="2"> [...] </tr>
```

1.7 Customizing headers and footers

By default an header and no footer will be rendered.

1.7.1 Adding column headers

The header cell for each column comes from `header`. By default this method returns `verbose_name`, falling back to the capitalized attribute name of the column in the table class.

When using QuerySet data and a verbose name has not been explicitly defined for a column, the corresponding model field's `verbose_name` will be used.

Consider the following:

```
>>> class Region(models.Model):
...     name = models.CharField(max_length=200)
...
>>> class Person(models.Model):
...     first_name = models.CharField(verbose_name="model verbose name", max_
↪length=200)
...     last_name = models.CharField(max_length=200)
...     region = models.ForeignKey('Region')
...
>>> class PersonTable(tables.Table):
...     first_name = tables.Column()
...     ln = tables.Column(accessor="last_name")
...     region_name = tables.Column(accessor="region__name")
...
>>> table = PersonTable(Person.objects.all())
>>> table.columns["first_name"].header
'Model Verbose Name'
>>> table.columns["ln"].header
'Last Name'
>>> table.columns["region_name"].header
'Name'
```

As you can see in the last example (region name), the results are not always desirable when an accessor is used to cross relationships. To get around this be careful to define `Column.verbose_name`.

Changing class names for ordered column headers

When a column is ordered in an ascending state there needs to be a way to show it in the interface. django-tables2 does this by adding an `asc` class for ascending or a `desc` class for descending. It should also be known that any orderable column is added with an `orderable` class to the column header.

Sometimes there may be a need to change these default classes.

On the `attrs` attribute of the table, you can add a `th` key with the value of a dictionary. Within that `th` dictionary, you may add an `_ordering` key also with the value of a dictionary.

The `_ordering` element is optional and all elements within it are optional. Inside you can have an `orderable` element, which will change the default `orderable` class name. You can also have `ascending` which will will change the default `asc` class name. And lastly, you can have `descending` which will change the default `desc` class name.

Example:

```
class Table(tables.Table):
    Meta:
        attrs = {
            "th" : {
                "_ordering": {
                    "orderable": "sortable", # Instead of `orderable`
                    "ascending": "ascend",   # Instead of `asc`
                    "descending": "descend"  # Instead of `desc`
                }
            }
        }
```

It can also be specified at initialization using the `attrs` for both: table and column:

```
ATTRIBUTES = {
    "th" : {
        "_ordering": {
            "orderable": "sortable", # Instead of `orderable`
            "ascending": "ascend",   # Instead of `asc`
            "descending": "descend"  # Instead of `desc`
        }
    }
}

table = tables.Table(queryset, attrs=ATTRIBUTES)

# or

class Table(tables.Table):
    my_column = tables.Column(attrs=ATTRIBUTES)
```

1.7.2 Adding column footers

By default, no footer will be rendered. If you want to add a footer, define a footer on at least one column.

That will make the table render a footer on every view of the table. It is up to you to decide if that makes sense if your table is paginated.

Pass `footer`-argument to the `Column` constructor.

The simplest case is just passing a `str` as the footer argument to a column:

```
country = tables.Column(footer="Total:")
```

This will just render the string in the footer. If you need to do more complex things, like showing a sum or an average, you can pass a callable:

```
population = tables.Column(
    footer=lambda table: sum(x["population"] for x in table.data)
)
```

You can expect `table`, `column` and `bound_column` as argument.

Define `render_footer` on a custom column.

If you need the same footer in multiple columns, you can create your own custom column. For example this column that renders the sum of the values in the column:

```
class SummingColumn(tables.Column):
    def render_footer(self, bound_column, table):
        return sum(bound_column.accessor.resolve(row) for row in table.data)
```

Then use this column like so:

```
class Table(tables.Table):
    name = tables.Column()
    country = tables.Column(footer="Total:")
    population = SummingColumn()
```

Note: If you are summing over tables with big datasets, chances are it is going to be slow. You should use some database aggregation function instead.

1.8 Swapping the position of columns

By default columns are positioned in the same order as they are declared, however when mixing auto-generated columns (via `Table.Meta.model`) with manually declared columns, the column sequence becomes ambiguous.

To resolve the ambiguity, columns sequence can be declared via the `Table.Meta.sequence` option:

```
class PersonTable(tables.Table):
    selection = tables.CheckBoxColumn(accessor="pk", orderable=False)

    class Meta:
        model = Person
        sequence = ('selection', 'first_name', 'last_name')
```

The special value `'...'` can be used to indicate that any omitted columns should inserted at that location. As such it can be used at most once.

1.9 Pagination

Pagination is easy, just call `Table.paginate()` and pass in the current page number:

```
def people_listing(request):
    table = PeopleTable(Person.objects.all())
    table.paginate(page=request.GET.get("page", 1), per_page=25)
    return render(request, "people_listing.html", {"table": table})
```

If you are using *`RequestConfig`*, pass pagination options to the constructor:

```
def people_listing(request):
    table = PeopleTable(Person.objects.all())
    RequestConfig(request, paginate={"per_page": 25}).configure(table)
    return render(request, "people_listing.html", {"table": table})
```

If you are using `SingleTableView`, the table will get paginated by default:

```
class PeopleListView(SingleTableView):
    table = PeopleTable
```

1.9.1 Disabling pagination

If you are using `SingleTableView` and want to disable the default behavior, set `SingleTableView.table_pagination = False`

1.9.2 Lazy pagination

The default `Paginator` wants to count the number of items, which might be an expensive operation for large QuerySets. In those cases, you can use *`LazyPaginator`*, which does not perform a count, but also does not know what the total amount of pages will be, until you've hit the last page.

The *`LazyPaginator`* does this by fetching `n + 1` records where the number of records per page is `n`. If it receives `n` or less records, it knows it is on the last page, preventing rendering of the 'next' button and further "…" ellipsis. Usage with `SingleTableView`:

```
class UserListView(SingleTableView):
    table_class = UserTable
    table_data = User.objects.all()
    paginator_class = LazyPaginator
```

1.10 Table Mixins

It's possible to create a mixin for a table that overrides something, however unless it itself is a subclass of `Table` class variable instances of *`Column`* will **not** be added to the class which is using the mixin.

Example:

```
>>> class UselessMixin:
...     extra = tables.Column()
...
>>> class TestTable(UselessMixin, tables.Table):
...     name = tables.Column()
...
>>> TestTable.base_columns.keys()
["name"]
```

To have a mixin contribute a column, it needs to be a subclass of `Table`. With this in mind the previous example *should* have been written as follows:

```
>>> class UsefulMixin(tables.Table):
...     extra = tables.Column()
...
>>> class TestTable(UsefulMixin, tables.Table):
...     name = tables.Column()
...
>>> TestTable.base_columns.keys()
["extra", "name"]
```

1.11 Customizing table style

1.11.1 CSS

In order to use CSS to style a table, you'll probably want to add a `class` or `id` attribute to the `<table>` element. django-tables2 has a hook that allows arbitrary attributes to be added to the `<table>` tag.

```
>>> import django_tables2 as tables
>>>
>>> class SimpleTable(tables.Table):
...     id = tables.Column()
...     age = tables.Column()
...
...     class Meta:
...         attrs = {"class": "mytable"}
...
>>> table = SimpleTable()
>>> # renders to something like this:
'<table class="mytable">...'
```

You can also specify `attrs` attribute when creating a column. `attrs` is a dictionary which contains attributes which by default get rendered on various tags involved with rendering a column. You can read more about them in *Column and row attributes*. django-tables2 supports three different dictionaries, this way you can give different attributes to column tags in table header (`th`), rows (`td`) or footer (`tf`)

```
>>> import django_tables2 as tables
>>>
>>> class SimpleTable(tables.Table):
...     id = tables.Column(attrs={"td": {"class": "my-class"}})
...     age = tables.Column(attrs={"tf": {"bgcolor": "red"}})
...
>>> table = SimpleTable()
>>> # renders to something like this:
'<tbody><tr><td class="my-class">...</td></tr>'
>>> # and the footer will look like this:
'<tfoot><tr> ... <td class="age" bgcolor="red"></tr></tfoot>''
```

1.11.2 Available templates

We ship a couple of different templates:

Template name	Description
django_tables2/table.html	Basic table template (default).
django_tables2/bootstrap.html	Template using bootstrap 3 structure/classes
django_tables2/bootstrap4.html	Template using bootstrap 4 structure/classes
django_tables2/bootstrap-responsive.html	Same as bootstrap, but wrapped in `.table-responsive`
django_tables2/semantic.html	Template using semantic UI

By default, django-tables2 looks for the `DJANGO_TABLES2_TEMPLATE` setting which is `django_tables2/table.html` by default.

If you use bootstrap 3 for your site, it makes sense to set the default to the bootstrap 3 template:

```
DJANGO_TABLES2_TEMPLATE = "django_tables2/bootstrap.html"
```

If you want to specify a custom template for selected tables in your project, you can set a `template_name` attribute to your custom `Table.Meta` class:

```
class PersonTable(tables.Table):

    class Meta:
        model = Person
        template_name = "django_tables2/semantic.html"
```

You can also use the `template_name` argument to the `Table` constructor to override the template for a certain instance:

```
table = PersonTable(data, template_name="django_tables2/bootstrap-responsive.html")
```

For none of the templates any CSS file is added to the HTML. You are responsible for including the relevant style sheets for a template.

1.11.3 Custom Template

And of course if you want full control over the way the table is rendered, ignore the built-in generation tools, and instead pass an instance of your `Table` subclass into your own template, and render it yourself.

You should use one of the provided templates as a basis.

1.12 Query string fields

Tables pass data via the query string to indicate ordering and pagination preferences.

The names of the query string variables are configurable via the options:

- `order_by_field` – default: `'sort'`
- `page_field` – default: `"page"`
- `per_page_field` – default: `"per_page"`, **note:** this field currently is not used by `{% render_table %}`

Each of these can be specified in three places:

- `Table.Meta.foo`
- `Table(..., foo=...)`
- `Table(...).foo = ...`

If you are using multiple tables on a single page, you will want to prefix these fields with a table-specific name, in order to prevent links on one table interfere with those on another table:

```
def people_listing(request):
    config = RequestConfig(request)
    table1 = PeopleTable(Person.objects.all(), prefix="1-")  # prefix specified
    table2 = PeopleTable(Person.objects.all(), prefix="2-")  # prefix specified
    config.configure(table1)
    config.configure(table2)
```

(continues on next page)

(continued from previous page)

```
    return render(request, 'people_listing.html', {
        'table1': table1,
        'table2': table2
    })
```

1.13 Controlling localization

Django-tables2 allows you to define which column of a table should or should not be localized. For example you may want to use this feature in following use cases:

- You want to format some columns representing for example numeric values in the given locales even if you don't enable `USE_L10N` in your settings file.
- You don't want to format primary key values in your table even if you enabled `USE_L10N` in your settings file.

This control is done by using two filter functions in Django's `l10n` library named `localize` and `unlocalize`. Check out Django docs about `localization` for more information about them.

There are two ways of controlling localization in your columns.

First one is setting the `localize` attribute in your column definition to `True` or `False`. Like so:

```
class PersonTable(tables.Table):
   id = tables.Column(accessor="pk", localize=False)
   class Meta:
       model = Person
```

Note: The default value of the `localize` attribute is `None` which means the formatting of columns is depending on the `USE_L10N` setting.

The second way is to define a `localize` and/or `unlocalize` tuples in your tables Meta class (like with `fields` or `exclude`). You can do this like so:

```
class PersonTable(tables.Table):
   id = tables.Column(accessor='pk')
   value = tables.Column(accessor='some_numerical_field')
   class Meta:
       model = Person
       unlocalize = ("id", )
       localize = ("value", )
```

If you define the same column in both `localize` and `unlocalize` then the value of this column will be 'unlocalized' which means that `unlocalize` has higher precedence.

1.14 Class Based Generic Mixins

Django-tables2 comes with two class based view mixins: `SingleTableMixin` and `MultiTableMixin`.

1.14.1 A single table using `SingleTableMixin`

`SingleTableMixin` makes it trivial to incorporate a table into a view or template.

The following view parameters are supported:

- `table_class` –- the table class to use, e.g. `SimpleTable`, if not specified and `model` is provided, a default table will be created on-the-fly.
- `table_data` (or `get_table_data()`) – the data used to populate the table
- `context_table_name` – the name of template variable containing the table object
- `table_pagination` (or `get_table_pagination`) – pagination options to pass to *`RequestConfig`*. Set `table_pagination=False` to disable pagination.
- **`get_table_kwargs()` allows the keyword arguments passed to the `Table` constructor.**

For example:

```
from django_tables2 import SingleTableView


class Person(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)


class PersonTable(tables.Table):
    class Meta:
        model = Person


class PersonList(SingleTableView):
    model = Person
    table_class = PersonTable
```

The template could then be as simple as:

```
{% load django_tables2 %}
{% render_table table %}
```

Such little code is possible due to the example above taking advantage of default values and `SingleTableMixin`'s eagerness at finding data sources when one is not explicitly defined.

Note: You don't have to base your view on `ListView`, you're able to mix `SingleTableMixin` directly.

1.14.2 Multiple tables using `MultiTableMixin`

If you need more than one table in a single view you can use `MultiTableMixin`. It manages multiple tables for you and takes care of adding the appropriate prefixes for them. Just define a list of tables in the tables attribute:

```
from django_tables2 import MultiTableMixin
from django.views.generic.base import TemplateView

class PersonTablesView(MultiTableMixin, TemplateView):
    template_name = "multiTable.html"
    tables = [
        PersonTable(qs),
        PersonTable(qs, exclude=("country", ))
    ]
```

(continues on next page)

(continued from previous page)

```
    table_pagination = {
        "per_page": 10
    }
```

In the template, you get a variable `tables`, which you can loop over like this:

```
{% load django_tables2 %}
{% for table in tables %}
    {% render_table table %}
{% endfor %}
```

1.15 Pinned rows

This feature allows one to pin certain rows to the top or bottom of your table. Provide an implementation for one or two of these methods, returning an iterable (QuerySet, list of dicts, list objects) representing the pinned data:

- `get_top_pinned_data(self)` - Displays the returned rows on top.
- `get_bottom_pinned_data(self)` - Displays the returned rows at the bottom.

Pinned rows are not affected by sorting and pagination, they will be present on every page of the table, regardless of ordering. Values will be rendered just like you are used to for normal rows.

Example:

```
class Table(tables.Table):
    first_name = tables.Column()
    last_name = tables.Column()

    def get_top_pinned_data(self):
        return [
            {"first_name": "Janet", "last_name": "Crossen"},
            # key "last_name" is None here, so the default value will be rendered.
            {"first_name": "Trine", "last_name": None}
        ]
```

Note: If you need very different rendering for the bottom pinned rows, chances are you actually want to use column footers: *Adding column footers*

1.15.1 Attributes for pinned rows

You can override the attributes used to render the `<tr>` tag of the pinned rows using: `pinned_row_attrs`. This works exactly like *Row attributes*.

Note: By default the `<tr>` tags for pinned rows will get the attribute `class="pinned-row"`.

```
<tr class="odd pinned-row" ...> [...] </tr>
<tr class="even pinned-row" ...> [...] </tr>
```

1.16 Filtering data in your table

When presenting a large amount of data, filtering is often a necessity. Fortunately, filtering the data in your django-tables2 table is simple with django-filter.

The basis of a filtered table is a `SingleTableMixin` combined with a `FilterView` from django-filter:

```
from django_filters.views import FilterView
from django_tables2.views import SingleTableMixin


class FilteredPersonListView(SingleTableMixin, FilterView):
    table_class = PersonTable
    model = Person
    template_name = "template.html"

    filterset_class = PersonFilter
```

The `FilterSet` is added to the template context in a `filter` variable by default. A basic template rendering the filter (using django-bootstrap3) and table looks like this:

```
{% load render_table from django_tables2 %}
{% load bootstrap3 %}

{% if filter %}
    <form action="" method="get" class="form form-inline">
        {% bootstrap_form filter.form layout='inline' %}
        {% bootstrap_button 'filter' %}
    </form>
{% endif %}
{% render_table table 'django_tables2/bootstrap.html' %}
```

1.17 Exporting table data

New in version 1.8.0.

If you want to allow exporting the data present in your django-tables2 tables to various formats, you must install the tablib package:

```
pip install tablib
```

Adding ability to export the table data to a class based views looks like this:

```
import django_tables2 as tables
from django_tables2.export.views import ExportMixin

from .models import Person
from .tables import MyTable

class TableView(ExportMixin, tables.SingleTableView):
    table_class = MyTable
    model = Person
    template_name = "django_tables2/bootstrap.html"
```

Now, if you append `_export=csv` to the query string, the browser will download a csv file containing your data. Supported export formats are:

csv, json, latex, ods, tsv, xls, xlsx, yaml

To customize the name of the query parameter add an `export_trigger_param` attribute to your class.

By default, the file will be named `table.ext`, where `ext` is the requested export format extension. To customize this name, add a `export_name` attribute to your class. The correct extension will be appended automatically to this value.

If you must use a function view, you might use something like this:

```
from django_tables2.config import RequestConfig
from django_tables2.export.export import TableExport

from .models import Person
from .tables import MyTable

def table_view(request):
    table = MyTable(Person.objects.all())

    RequestConfig(request).configure(table)

    export_format = request.GET.get("_export", None)
    if TableExport.is_valid_format(export_format):
        exporter = TableExport(export_format, table)
        return exporter.response("table.{}".format(export_format))

    return render(request, "table.html", {
        "table": table
    })
```

1.17.1 What exactly is exported?

The export views use the `Table.as_values()` method to get the data from the table. Because we often use HTML in our table cells, we need to specify something else for the export to make sense.

If you use *Table.render_foo methods*-methods to customize the output for a column, you should define a *Table.value_foo methods*-method, returning the value you want to be exported.

If you are creating your own custom columns, you should know that each column defines a `value()` method, which is used in `Table.as_values()`. By default, it just calls the `render()` method on that column. If your custom column produces HTML, you should override this method and return the actual value.

1.17.2 Including and excluding columns

Some data might be rendered in the HTML version of the table using color coding, but need a different representation in an export format. Use columns with `visible=False` to include columns in the export, but not visible in the regular rendering:

```
class Table(tables.Table):
    name = columns.Column(exclude_from_export=True)
    first_name = columns.Column(visible=False)
    last_name = columns.Column(visible=False)
```

Certain columns do not make sense while exporting data: you might show images or have a column with buttons you want to exclude from the export. You can define the columns you want to exclude in several ways:

```
# exclude a column while defining Columns on a table:
class Table(tables.Table):
    name = columns.Column()
    buttons = columns.TemplateColumn(template_name="...", exclude_from_export=True)


# exclude columns while creating the TableExport instance:
exporter = TableExport("csv", table, exclude_columns=("image", "buttons"))
```

If you use the `django_tables2.export.ExportMixin`, add an `exclude_columns` attribute to your class:

```
class TableView(ExportMixin, tables.SingleTableView):
    table_class = MyTable
    model = Person
    template_name = 'django_tables2/bootstrap.html'
    exclude_columns = ("buttons", )
```

1.17.3 Tablib Dataset Configuration

django-tables2 uses `tablib` to export the table data. You may pass kwargs to the `tablib.Dataset` via the `TableExport` constructor `dataset_kwargs` parameter:

```
exporter = TableExport("xlsx", table, dataset_kwargs={"title": "My Custom Sheet Name"}
↪)
```

Default for `tablib.Dataset.title` is based on `table.Meta.model._meta.verbose_name_plural.title()`, if available.

If you use the `django_tables2.export.ExportMixin`, simply add a `dataset_kwargs` attribute to your class:

```
class View(ExportMixin, tables.SingleTableView):
    table_class = MyTable
    model = Person
    dataset_kwargs = {"title": "People"}
```

or override the `ExportMixin.get_dataset_kwargs` method to return the kwargs dictionary dynamically.

1.17.4 Generating export URLs

You can use the `export_url` template tag included with django_tables2 to render a link to export the data as `csv`:

```
{% export_url "csv" %}
```

This will make sure any other query string parameters will be preserved, for example in combination when filtering table items.

If you want to render more than one button, you could use something like this:

```
{% for format in view.export_formats %}
    <a href="{% export_url format %}">
        download  <code>.{{ format }}</code>
    </a>
{% endfor %}
```

Note: This example assumes you define a list of possible export formats on your view instance in attribute `export_formats`.

1.18 API

1.18.1 Built-in columns

For common use-cases the following columns are included:

- *BooleanColumn* – renders boolean values
- *Column* – generic column
- *CheckBoxColumn* – renders `checkbox` form inputs
- *DateColumn* – date formatting
- *DateTimeColumn* – `datetime` formatting in the local timezone
- *EmailColumn* – renders `<a href="mailto:...">` tags
- *FileColumn* – renders files as links
- *JSONColumn* – renders JSON as an indented string in `<pre></pre>`
- *LinkColumn* – renders `<a href="...">` tags (compose a Django URL)
- *ManyToManyColumn* – renders a list objects from a `ManyToManyField`
- *RelatedLinkColumn* – renders `<a href="...">` tags linking related objects
- *TemplateColumn* – renders template code
- *URLColumn* – renders `<a href="...">` tags (absolute URL)

1.18.2 Template tags

render_table

Renders a `Table` object to HTML and enables as many features in the output as possible.

```
{% load django_tables2 %}
{% render_table table %}

{# Alternatively a specific template can be used #}
{% render_table table "path/to/custom_table_template.html" %}
```

If the second argument (template path) is given, the template will be rendered with a `RequestContext` and the table will be in the variable `table`.

Note: This tag temporarily modifies the `Table` object during rendering. A `context` attribute is added to the table, providing columns with access to the current context for their own rendering (e.g. *TemplateColumn*).

This tag requires that the template in which it's rendered contains the `HttpRequest` inside a `request` variable. This can be achieved by ensuring the `TEMPLATES[][‘OPTIONS’][‘context_processors’]` setting contains `django.template.context_processors.request`. Please refer to the Django documentation for the TEMPLATES-setting.

querystring

A utility that allows you to update a portion of the query-string without overwriting the entire thing.

Let's assume we have the query string `?search=pirates&sort=name&page=5` and we want to update the `sort` parameter:

```
{% querystring "sort"="dob" %}            # ?search=pirates&sort=dob&page=5
{% querystring "sort"="" %}               # ?search=pirates&page=5
{% querystring "sort"="" "search"="" %}   # ?page=5

{% with "search" as key %}                # supports variables as keys
{% querystring key="robots" %}            # ?search=robots&page=5
{% endwith %}
```

This tag requires the `django.template.context_processors.request` context processor, see *render_table*.

1.18.3 API Reference

Accessor (A)

RequestConfig

class django_tables2.config.**RequestConfig**(*request*, *paginate=True*)

A configurator that uses request data to setup a table.

A single RequestConfig can be used for multiple tables in one view.

Parameters **paginate** (*dict or bool*) – Indicates whether to paginate, and if so, what default values to use. If the value evaluates to `False`, pagination will be disabled. A `dict` can be used to specify default values for the call to `paginate` (e.g. to define a default `per_page` value).

A special *silent* item can be used to enable automatic handling of pagination exceptions using the following logic:

- If `PageNotAnInteger` is raised, show the first page.
- If `EmptyPage` is raised, show the last page.

For example, to use *LazyPaginator*:

```
RequestConfig(paginate={"paginator_class": LazyPaginator}).
↪configure(table)
```

Table

Table.Meta

class Table.**Meta**

Provides a way to define *global* settings for table, as opposed to defining them for each instance.

For example, if you want to create a table of users with their primary key added as a `data-id` attribute on each `<tr>`, You can use the following:

```
class UsersTable(tables.Table):
    class Meta:
        row_attrs = {"data-id": lambda record: record.pk}
```

Which adds the desired `row_attrs` to every instance of `UsersTable`, in contrast of defining it at construction time:

```
table = tables.Table(User.objects.all(),
                     row_attrs={"data-id": lambda record: record.pk})
```

Some settings are only available in *`Table.Meta`* and not as an argument to the `Table` constructor.

Note: If you define a `class Meta` on a child of a table already having a `class Meta` defined, you need to specify the parent's `Meta` class as the parent for the `class Meta` in the child:

```
class PersonTable(table.Table):
    class Meta:
        model = Person
        exclude = ("email", )

class PersonWithEmailTable(PersonTable):
    class Meta(PersonTable.Meta):
        exclude = ()
```

All attributes are overwritten if defined in the child's `class Meta`, no merging is attempted.

Arguments:

attrs (`dict`): Add custom HTML attributes to the table. Allows custom HTML attributes to be specified which will be added to the `<table>` tag of any table rendered via `Table.as_html()` or the *render_table* template tag.

This is typically used to enable a theme for a table (which is done by adding a CSS class to the `<table>` element):

```
class SimpleTable(tables.Table):
    name = tables.Column()

    class Meta:
        attrs = {"class": "paleblue"}
```

If you supply a a callable as a value in the dict, it will be called at table instantiation an the returned value will be used:

Consider this example where each table gets an unique `"id"` attribute:

```
import itertools
counter = itertools.count()

class UniqueIdTable(tables.Table):
    name = tables.Column()

    class Meta:
        attrs = {"id": lambda: "table_{}".format(next(counter))}
```

Note: This functionality is also available via the `attrs` keyword argument to a table's constructor.

row_attrs (`dict`): Add custom html attributes to the table rows. Allows custom HTML attributes to be specified which will be added to the `<tr>` tag of the rendered table. Optional keyword arguments are `table` and `record`.

This can be used to add each record's primary key to each row:

```
class PersonTable(tables.Table):
    class Meta:
        model = Person
        row_attrs = {"data-id": lambda record: record.pk}

# will result in
'<tr data-id="1">...</tr>'
```

Note: This functionality is also available via the `row_attrs` keyword argument to a table's constructor.

empty_text (str): Defines the text to display when the table has no rows. If the table is empty and `bool(empty_text)` is `True`, a row is displayed containing `empty_text`. This is allows a message such as *There are currently no FOO.* to be displayed.

Note: This functionality is also available via the `empty_text` keyword argument to a table's constructor.

show_header (bool): Whether or not to show the table header. Defines whether the table header should be displayed or not, by default, the header shows the column names.

Note: This functionality is also available via the `show_header` keyword argument to a table's constructor.

exclude (tuple): Exclude columns from the table. This is useful in subclasses to exclude columns in a parent:

```
>>> class Person(tables.Table):
...     first_name = tables.Column()
...     last_name = tables.Column()
...
>>> Person.base_columns
{'first_name': <django_tables2.columns.Column object at 0x10046df10>,
'last_name': <django_tables2.columns.Column object at 0x10046d8d0>}
>>> class ForgetfulPerson(Person):
...     class Meta:
...         exclude = ("last_name", )
...
>>> ForgetfulPerson.base_columns
{'first_name': <django_tables2.columns.Column object at 0x10046df10>}
```

Note: This functionality is also available via the `exclude` keyword argument to a table's construc-

tor.

However, unlike some of the other *Table.Meta* options, providing the `exclude` keyword to a table's constructor **won't override** the `Meta.exclude`. Instead, it will be effectively be *added* to it. i.e. you can't use the constructor's `exclude` argument to *undo* an exclusion.

fields (`tuple`): Fields to show in the table. Used in conjunction with `model`, specifies which fields should have columns in the table. If `None`, all fields are used, otherwise only those named:

```
# models.py
class Person(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)

# tables.py
class PersonTable(tables.Table):
    class Meta:
        model = Person
        fields = ("first_name", )
```

model (`django.core.db.models.Model`): Create columns from model. A model to inspect and automatically create corresponding columns.

This option allows a Django model to be specified to cause the table to automatically generate columns that correspond to the fields in a model.

order_by (tuple or str): The default ordering tuple or comma separated str. A hyphen `-` can be used to prefix a column name to indicate *descending* order, for example: `('name', '-age')` or `name,-age`.

Note: This functionality is also available via the `order_by` keyword argument to a table's constructor.

---

sequence (iterable): The sequence of the table columns. This allows the default order of columns (the order they were defined in the Table) to be overridden.

The special item `'...'` can be used as a placeholder that will be replaced with all the columns that were not explicitly listed. This allows you to add columns to the front or back when using inheritance.

Example:

```
>>> class Person(tables.Table):
...     first_name = tables.Column()
...     last_name = tables.Column()
...
...     class Meta:
...         sequence = ("last_name", "...")
...
>>> Person.base_columns.keys()
['last_name', 'first_name']
```

The `'...'` item can be used at most once in the sequence value. If it is not used, every column *must* be explicitly included. For example in the above example, `sequence = ('last_name', )` would be **invalid** because neither `"..."` or `"first_name"` were included.

Note: This functionality is also available via the `sequence` keyword argument to a table's constructor.

---

orderable (bool): Default value for column's *orderable* attribute. If the table and column don't specify a value, a column's `orderable` value will fall back to this. This provides an easy mechanism to disable ordering on an entire table, without adding `orderable=False` to each column in a table.

Note: This functionality is also available via the `orderable` keyword argument to a table's constructor.

---

template_name (str): The name of template to use when rendering the table.

Note: This functionality is also available via the `template_name` keyword argument to a table's constructor.

---

localize (tuple): Specifies which fields should be localized in the table. Read *Controlling localization* for more information.

unlocalize (tuple): Specifies which fields should be unlocalized in the table. Read *Controlling localization* for more information.

Columns

Column

class django_tables2.columns.**Column**(*verbose_name=None*, *accessor=None*, *default=None*, *visible=True*, *orderable=None*, *attrs=None*, *order_by=None*, *empty_values=None*, *localize=None*, *footer=None*, *exclude_from_export=False*, *linkify=False*, *initial_sort_descending=False*)

Represents a single column of a table.

Column objects control the way a column (including the cells that fall within it) are rendered.

Parameters

- **attrs** (*dict*) – HTML attributes for elements that make up the column. This API is extended by subclasses to allow arbitrary HTML attributes to be added to the output.

  By default *Column* supports:

  – `th` – `table/thead/tr/th` elements

  – `td` – `table/tbody/tr/td` elements

  – `cell` – fallback if `th` or `td` is not defined

  – `a` – To control the attributes for the `a` tag if the cell is wrapped in a link.

- **accessor** (str or `Accessor`) – An accessor that describes how to extract values for this column from the *table data*.

- **default** (*str or callable*) – The default value for the column. This can be a value or a callable object[1]. If an object in the data provides `None` for a column, the default will be used instead.

  The default value may affect ordering, depending on the type of data the table is using. The only case where ordering is not affected is when a `QuerySet` is used as the table data (since sorting is performed by the database).

[1] The provided callable object must not expect to receive any arguments.

- **empty_values** (*iterable*) – list of values considered as a missing value, for which the column will render the default value. Defaults to `(None, '')`
- **exclude_from_export** (*bool*) – If `True`, this column will not be added to the data iterator returned from as_values().
- **footer** (*str, callable*) – Defines the footer of this column. If a callable is passed, it can take optional keyword arguments `column`, `bound_column` and `table`.
- **order_by** (str, tuple or `Accessor`) – Allows one or more accessors to be used for ordering rather than *accessor*.
- **orderable** (*bool*) – If `False`, this column will not be allowed to influence row ordering/sorting.
- **verbose_name** (*str*) – A human readable version of the column name.
- **visible** (*bool*) – If `True`, this column will be rendered. Columns with `visible=False` will not be rendered, but will be included in `.Table.as_values()` and thus also in *Exporting table data*.
- **localize** – If the cells in this column will be localized by the `localize` filter:
  - If `True`, force localization
  - If `False`, values are not localized
  - If `None` (default), localization depends on the `USE_L10N` setting.
- **linkify** (*bool, str, callable, dict, tuple*) – Controls if cell content will be wrapped in an `a` tag. The different ways to define the `href` attribute:
  - If `True`, the `record.get_absolute_url()` or the related model's `get_absolute_url()` is used.
  - If a callable is passed, the returned value is used, if it's not `None`. The callable can optionally accept any argument valid for *Table.render_foo methods*-methods, for example `record` or *value*.
  - If a `dict` is passed, it's passed on to `~django.urls.reverse`.
  - If a `tuple` is passed, it must be either a (viewname, args) or (viewname, kwargs) tuple, which is also passed to `~django.urls.reverse`.

Examples, assuming this model:

```
class Blog(models.Model):
    title = models.CharField(max_length=100)
    body = model.TextField()
    user = model.ForeignKey(get_user_model(), on_delete=models.CASCADE)
```

Using the `linkify` argument to control the linkification. These columns will all display the value returned from `str(record.user)`:

```
# If the column is named 'user', the column will use record.user.get_absolute_
↪url()
user = tables.Column(linkify=True)

# We can also do that explicitly:
user = tables.Column(linkify=lambda record: record.user.get_absolute_url())

# or, if no get_absolute_url is defined, or a custom link is required, we have a␣
↪couple
```

(continues on next page)

(continued from previous page)

```
# of ways to define what is passed to reverse()
user = tables.Column(linkify={"viewname": "user_detail", "args": [tables.A("user__
↪pk")]})
user = tables.Column(linkify=("user_detail", [tables.A("user__pk")])) # (viewname,
↪ args)
user = tables.Column(linkify=("user_detail", {"pk": tables.A("user__pk")})) #␣
↪(viewname, kwargs)

initial_sort_descending (bool): If `True`, a column will sort in descending order
    on "first click" after table has been rendered. If `False`, column will follow
    default behavior, and sort ascending on "first click". Defaults to `False`.
```

order (*queryset*, *is_descending*)
Returns the QuerySet of the table.

This method can be overridden by *table.order_FOO() methods* methods on the table or by subclassing `Column`; but only overrides if second element in return tuple is True.

Returns Tuple (QuerySet, boolean)

render (*value*)
Returns the content for a specific cell.

This method can be overridden by *Table.render_foo methods* methods on the table or by subclassing `Column`.

If the value for this cell is in `empty_values`, this method is skipped and an appropriate default value is rendered instead. Subclasses should set `empty_values` to `()` if they want to handle all values in `render`.

value (***kwargs*)
Returns the content for a specific cell similarly to `render` however without any html content. This can be used to get the data in the formatted as it is presented but in a form that could be added to a csv file.

The default implementation just calls the `render` function but any subclasses where `render` returns html content should override this method.

See `LinkColumn` for an example.

BooleanColumn

class django_tables2.columns.**BooleanColumn** (*null=False*, *yesno='✓, '*, ***kwargs*)
A column suitable for rendering boolean data.

Parameters

- **null** (*bool*) – is `None` different from `False`?
- **yesno** (*str*) – comma separated values string or 2-tuple to display for True/False values.

Rendered values are wrapped in a `<span>` to allow customization by using CSS. By default the span is given the class `true`, `false`.

In addition to *attrs* keys supported by `Column`, the following are available:

- `span` – adds attributes to the `<span>` tag

CheckBoxColumn

class django_tables2.columns.**CheckBoxColumn**(*attrs=None*, *checked=None*, ***extra*)

A subclass of *Column* that renders as a checkbox form input.

This column allows a user to *select* a set of rows. The selection information can then be used to apply some operation (e.g. "delete") onto the set of objects that correspond to the selected rows.

The value that is extracted from the *table data* for this column is used as the value for the checkbox, i.e. `<input type="checkbox" value="..." />`

This class implements some sensible defaults:

- HTML input's `name` attribute is the *column name* (can override via *attrs* argument).
- `orderable` defaults to `False`.

Parameters

- **attrs** (*dict*) – In addition to *attrs* keys supported by *Column*, the following are available:
  - `input` – `<input>` elements in both `<td>` and `<th>`.
  - `th__input` – Replaces `input` attrs in header cells.
  - `td__input` – Replaces `input` attrs in body cells.
- **checked** (`Accessor`, bool, callable) – Allow rendering the checkbox as checked. If it resolves to a truthy value, the checkbox will be rendered as checked.

Note: You might expect that you could select multiple checkboxes in the rendered table and then *do something* with that. This functionality is not implemented. If you want something to actually happen, you will need to implement that yourself.

---

property header

The value used for the column heading (e.g. inside the `<th>` tag).

By default this returns `verbose_name`.

Returns `unicode` or `None`

Note: This property typically is not accessed directly when a table is rendered. Instead, *BoundColumn.header* is accessed which in turn accesses this property. This allows the header to fallback to the column name (it is only available on a *BoundColumn* object hence accessing that first) when this property doesn't return something useful.

---

is_checked(*value*, *record*)

Determine if the checkbox should be checked

render(*value*, *bound_column*, *record*)

Returns the content for a specific cell.

This method can be overridden by *Table.render_foo methods* methods on the table or by subclassing *Column*.

If the value for this cell is in `empty_values`, this method is skipped and an appropriate default value is rendered instead. Subclasses should set `empty_values` to `()` if they want to handle all values in *render*.

DateColumn

class django_tables2.columns.**DateColumn**(*format=None*, *short=True*, **args*, ***kwargs*)
A column that renders dates in the local timezone.

Parameters

- **format** (*str*) – format string in same format as Django's `date` template filter (optional)
- **short** (*bool*) – if `format` is not specified, use Django's `SHORT_DATE_FORMAT` setting, otherwise use `DATE_FORMAT`

classmethod from_field(*field*, ***kwargs*)
Return a specialized column for the model field or `None`.

Parameters **field** (*Model Field instance*) – the field that needs a suitable column

Returns *Column* object or `None`

If the column is not specialized for the given model field, it should return `None`. This gives other columns the opportunity to do better.

If the column is specialized, it should return an instance of itself that is configured appropriately for the field.

DateTimeColumn

class django_tables2.columns.**DateTimeColumn**(*format=None*, *short=True*, **args*, ***kwargs*)
A column that renders `datetime` instances in the local timezone.

Parameters

- **format** (*str*) – format string for datetime (optional). Note that *format* uses Django's `date` template tag syntax.
- **short** (*bool*) – if `format` is not specified, use Django's `SHORT_DATETIME_FORMAT`, else `DATETIME_FORMAT`

classmethod from_field(*field*, ***kwargs*)
Return a specialized column for the model field or `None`.

Parameters **field** (*Model Field instance*) – the field that needs a suitable column

Returns *Column* object or `None`

If the column is not specialized for the given model field, it should return `None`. This gives other columns the opportunity to do better.

If the column is specialized, it should return an instance of itself that is configured appropriately for the field.

EmailColumn

class django_tables2.columns.**EmailColumn**(*text=None*, **args*, ***kwargs*)
Render email addresses to `mailto:`-links.

Parameters

- **attrs** (*dict*) – HTML attributes that are added to the rendered `<a href="...">...</a>` tag.

- **text** – Either static text, or a callable. If set, this will be used to render the text inside link instead of the value.

Example:

```
# models.py
class Person(models.Model):
    name = models.CharField(max_length=200)
    email =  models.EmailField()

# tables.py
class PeopleTable(tables.Table):
    name = tables.Column()
    email = tables.EmailColumn()

# result
# [...]<a href="mailto:email@example.com">email@example.com</a>
```

classmethod from_field(*field*, ***kwargs*)
Return a specialized column for the model field or `None`.

Parameters **field** (*Model Field instance*) – the field that needs a suitable column

Returns *Column* object or `None`

If the column is not specialized for the given model field, it should return `None`. This gives other columns the opportunity to do better.

If the column is specialized, it should return an instance of itself that is configured appropriately for the field.

FileColumn

class django_tables2.columns.**FileColumn**(*verify_exists=True*, ***kwargs*)
Attempts to render `FieldFile` (or other storage backend `File`) as a hyperlink.

When the file is accessible via a URL, the file is rendered as a hyperlink. The `basename` is used as the text, wrapped in a span:

```
<a href="/media/path/to/receipt.pdf" title="path/to/receipt.pdf">receipt.pdf</a>
```

When unable to determine the URL, a `span` is used instead:

```
<span title="path/to/receipt.pdf" class>receipt.pdf</span>
```

`Column.attrs` keys `a` and `span` can be used to add additional attributes.

Parameters

- **verify_exists** (*bool*) – attempt to determine if the file exists If *verify_exists*, the HTML class `exists` or `missing` is added to the element to indicate the integrity of the storage.
- **text** (*str or callable*) – Either static text, or a callable. If set, this will be used to render the text inside the link instead of the file's `basename` (default)

classmethod from_field(*field*, ***kwargs*)
Return a specialized column for the model field or `None`.

Parameters **field** (*Model Field instance*) – the field that needs a suitable column

Returns *Column* object or None

If the column is not specialized for the given model field, it should return None. This gives other columns the opportunity to do better.

If the column is specialized, it should return an instance of itself that is configured appropriately for the field.

render(*record*, *value*)
Returns the content for a specific cell.

This method can be overridden by *Table.render_foo methods* methods on the table or by subclassing *Column*.

If the value for this cell is in empty_values, this method is skipped and an appropriate default value is rendered instead. Subclasses should set empty_values to () if they want to handle all values in *render*.

JSONColumn

class django_tables2.columns.**JSONColumn**(*json_dumps_kwargs=None*, ***kwargs*)
Render the contents of JSONField or HStoreField as an indented string.

New in version 1.5.0.

Note: Automatic rendering of data to this column requires PostgreSQL support (psycopg2 installed) to import the fields, but this column can also be used manually without it.

Parameters

- **json_dumps_kwargs** – kwargs passed to json.dumps, defaults to {'indent': 2}
- **attrs** (*dict*) – In addition to *attrs* keys supported by *Column*, the following are available:
  - pre – <pre> around the rendered JSON string in <td> elements.

classmethod from_field(*field*, ***kwargs*)
Return a specialized column for the model field or None.

Parameters **field** (*Model Field instance*) – the field that needs a suitable column

Returns *Column* object or None

If the column is not specialized for the given model field, it should return None. This gives other columns the opportunity to do better.

If the column is specialized, it should return an instance of itself that is configured appropriately for the field.

render(*record*, *value*)
Returns the content for a specific cell.

This method can be overridden by *Table.render_foo methods* methods on the table or by subclassing *Column*.

If the value for this cell is in `empty_values`, this method is skipped and an appropriate default value is rendered instead. Subclasses should set `empty_values` to `()` if they want to handle all values in *render*.

LinkColumn

class django_tables2.columns.**LinkColumn**(*viewname=None*, *urlconf=None*, *args=None*, *kwargs=None*, *current_app=None*, *attrs=None*, ***extra*)

Renders a normal value as an internal hyperlink to another page.

Note: This column should not be used anymore, the `linkify` keyword argument to regular columns can be used to achieve the same results.

It's common to have the primary value in a row hyperlinked to the page dedicated to that record.

The first arguments are identical to that of `reverse` and allows an internal URL to be described. If this argument is `None`, then `get_absolute_url`. (see Django references) will be used. The last argument *attrs* allows custom HTML attributes to be added to the rendered `<a href="...">` tag.

Parameters

- **viewname** (*str or None*) – See `reverse`, or use `None` to use the model's `get_absolute_url`
- **urlconf** (*str*) – See `reverse`.
- **args** (*list*) – See `reverse`.[2]
- **kwargs** (*dict*) – See `reverse`.[2]
- **current_app** (*str*) – See `reverse`.
- **attrs** (*dict*) – HTML attributes that are added to the rendered `<a ...>...</a>` tag.
- **text** (*str or callable*) – Either static text, or a callable. If set, this will be used to render the text inside link instead of value (default). The callable gets the record being rendered as argument.

Example:

```
# models.py
class Person(models.Model):
    name = models.CharField(max_length=200)

# urls.py
urlpatterns = patterns('',
    url("people/([0-9]+)/", views.people_detail, name="people_detail")
)

# tables.py
from django_tables2.utils import A  # alias for Accessor

class PeopleTable(tables.Table):
    name = tables.LinkColumn("people_detail", args=[A("pk")])
```

[2] In order to create a link to a URL that relies on information in the current row, `Accessor` objects can be used in the *args* or *kwargs* arguments. The accessor will be resolved using the row's record before `reverse` is called.

In order to override the text value (i.e. `<a ... >text</a>`) consider the following example:

```
# tables.py
from django_tables2.utils import A  # alias for Accessor

class PeopleTable(tables.Table):
    name = tables.LinkColumn("people_detail", text="static text", args=[A("pk")])
    age  = tables.LinkColumn("people_detail", text=lambda record: record.name,␣
↪args=[A("pk")])
```

In the first example, a static text would be rendered (`"static text"`) In the second example, you can specify a callable which accepts a record object (and thus can return anything from it)

In addition to *attrs* keys supported by *Column*, the following are available:

- `a` – `<a>` elements in `<td>`.

Adding attributes to the `<a>`-tag looks like this:

```
class PeopleTable(tables.Table):
    first_name = tables.LinkColumn(attrs={
        "a": {"style": "color: red;"}
    })
```

ManyToManyColumn

class django_tables2.columns.**ManyToManyColumn**(*transform=None*, *filter=None*, *separator=', '*, *linkify_item=None*, **args*, ***kwargs*)

Display the list of objects from a `ManyRelatedManager`

Ordering is disabled for this column.

Parameters

- **transform** – callable to transform each item to text, it gets an item as argument and must return a string-like representation of the item. By default, it calls `force_str` on each item.
- **filter** – callable to filter, limit or order the QuerySet, it gets the `ManyRelatedManager` as first argument and must return a filtered QuerySet. By default, it returns `all()`
- **separator** – separator string to join the items with. default: `", "`
- **linkify_item** – callable, arguments to reverse() or `True` to wrap items in a `<a>` tag. For a detailed explanation, see `linkify` argument to `Column`.

For example, when displaying a list of friends with their full name:

```
# models.py
class Person(models.Model):
    first_name = models.CharField(max_length=200)
    last_name = models.CharField(max_length=200)
    friends = models.ManyToManyField(Person)
    is_active = models.BooleanField(default=True)

    @property
    def name(self):
        return '{} {}'.format(self.first_name, self.last_name)
```

(continues on next page)

(continued from previous page)

```
# tables.py
class PersonTable(tables.Table):
    name = tables.Column(order_by=("last_name", "first_name"))
    friends = tables.ManyToManyColumn(transform=lambda user: u.name)
```

If only the active friends should be displayed, you can use the *filter* argument:

```
friends = tables.ManyToManyColumn(filter=lambda qs: qs.filter(is_active=True))
```

filter(*qs*)
Filter is called on the ManyRelatedManager to allow ordering, filtering or limiting on the set of related objects.

classmethod from_field(*field*, ***kwargs*)
Return a specialized column for the model field or `None`.

Parameters **field** (*Model Field instance*) – the field that needs a suitable column

Returns *Column* object or `None`

If the column is not specialized for the given model field, it should return `None`. This gives other columns the opportunity to do better.

If the column is specialized, it should return an instance of itself that is configured appropriately for the field.

render(*value*)
Returns the content for a specific cell.

This method can be overridden by *Table.render_foo methods* methods on the table or by subclassing *Column*.

If the value for this cell is in `empty_values`, this method is skipped and an appropriate default value is rendered instead. Subclasses should set `empty_values` to `()` if they want to handle all values in *render*.

transform(*obj*)
Transform is applied to each item of the list of objects from the ManyToMany relation.

RelatedLinkColumn

class django_tables2.columns.**RelatedLinkColumn**(*viewname=None*, *urlconf=None*, *args=None*, *kwargs=None*, *current_app=None*, *attrs=None*, ***extra*)
Render a link to a related object using related object's `get_absolute_url`, same parameters as `~.LinkColumn`.

Note: This column should not be used anymore, the `linkify` keyword argument to regular columns can be used achieve the same results.

If the related object does not have a method called `get_absolute_url`, or if it is not callable, the link will be rendered as '#'.

Traversing relations is also supported, suppose a Person has a foreign key to Country which in turn has a foreign key to Continent:

```
class PersonTable(tables.Table):
    name = tables.Column()
    country = tables.RelatedLinkColumn()
    continent = tables.RelatedLinkColumn(accessor="country.continent")
```

will render:

- in column 'country', link to `person.country.get_absolute_url()` with the output of `str(person.country)` as <a> contents.
- in column 'continent', a link to `person.country.continent.get_absolute_url()` with the output of `str(person.country.continent)` as <a> contents.

Alternative contents of <a> can be supplied using the `text` keyword argument as documented for *LinkColumn*.

TemplateColumn

class django_tables2.columns.**TemplateColumn**(*template_code=None*, *template_name=None*, *extra_context=None*, ***extra*)

A subclass of *Column* that renders some template code to use as the cell value.

Parameters

- **template_code** (*str*) – template code to render
- **template_name** (*str*) – name of the template to render
- **extra_context** (*dict*) – optional extra template context

A `Template` object is created from the *template_code* or *template_name* and rendered with a context containing:

- *record* – data record for the current row
- *value* – value from `record` that corresponds to the current column
- *default* – appropriate default value to use as fallback.
- *row_counter* – The number of the row this cell is being rendered in.
- any context variables passed using the `extra_context` argument to *TemplateColumn*.

Example:

```
class ExampleTable(tables.Table):
    foo = tables.TemplateColumn("{{ record.bar }}")
    # contents of `myapp/bar_column.html` is `{{ label }}: {{ value }}`
    bar = tables.TemplateColumn(template_name="myapp/name2_column.html",
                                extra_context={"label": "Label"})
```

Both columns will have the same output.

render(*record*, *table*, *value*, *bound_column*, ***kwargs*)

Returns the content for a specific cell.

This method can be overridden by *Table.render_foo methods* methods on the table or by subclassing *Column*.

If the value for this cell is in `empty_values`, this method is skipped and an appropriate default value is rendered instead. Subclasses should set `empty_values` to `()` if they want to handle all values in *render*.

value (***kwargs*)
The value returned from a call to *value()* on a *TemplateColumn* is the rendered template with `django.utils.html.strip_tags` applied.

URLColumn

class django_tables2.columns.**URLColumn**(*text=None*, **args*, ***kwargs*)
Renders URL values as hyperlinks.

Parameters

- **text** (*str or callable*) – Either static text, or a callable. If set, this will be used to render the text inside link instead of value (default)
- **attrs** (*dict*) – Additional attributes for the <a> tag

Example:

```
>>> class CompaniesTable(tables.Table):
...     link = tables.URLColumn()
...
>>> table = CompaniesTable([{"link": "http://google.com"}])
>>> table.rows[0].get_cell("link")
'<a href="http://google.com">http://google.com</a>'
```

classmethod from_field(*field*, ***kwargs*)
Return a specialized column for the model field or `None`.

Parameters **field** (*Model Field instance*) – the field that needs a suitable column

Returns *Column* object or `None`

If the column is not specialized for the given model field, it should return `None`. This gives other columns the opportunity to do better.

If the column is specialized, it should return an instance of itself that is configured appropriately for the field.

Views, view mixins and paginators

SingleTableMixin

MultiTableMixin

SingleTableView

export.TableExport

export.ExportMixin

LazyPaginator

class django_tables2.paginators.**LazyPaginator**(*object_list*, *per_page*, *look_ahead=None*, ***kwargs*)

Implement lazy pagination, preventing any count() queries.

By default, for any valid page, the total number of pages for the paginator will be

- `current + 1` if the number of records fetched for the current page offset is bigger than the number of records per page.
- `current` if the number of records fetched is less than the number of records per page.

The number of additional records fetched can be adjusted using `look_ahead`, which defaults to 1 page. If you like to provide a little more extra information on how much pages follow the current page, you can use a higher value.

Note: The number of records fetched for each page is `per_page * look_ahead + 1`, so increasing the value for `look_ahead` makes the view a bit more expensive.

So:

```
paginator = LazyPaginator(range(10000), 10)

>>> paginator.page(1).object_list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> paginator.num_pages
2
>>> paginator.page(10).object_list
[91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
>>> paginator.num_pages
11
>>> paginator.page(1000).object_list
[9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
>>> paginator.num_pages
1000
```

Usage with `SingleTableView`:

```
class UserListView(SingleTableView):
    table_class = UserTable
    table_data = User.objects.all()
    pagination_class = LazyPaginator
```

Or with *`RequestConfig`*:

```
RequestConfig(paginate={"paginator_class": LazyPaginator}).configure(table)
```

New in version 2.0.0.

See *Internal APIs* for internal classes.

1.18.4 Internal APIs

The items documented here are internal and subject to change.

BoundColumns

class django_tables2.columns.**BoundColumns**(*table*, *base_columns*)
Container for spawning *BoundColumn* objects.

This is bound to a table and provides its `Table.columns` property. It provides access to those columns in different ways (iterator, item-based, filtered and unfiltered etc), stuff that would not be possible with a simple iterator in the table class.

A *BoundColumns* object is a container for holding *BoundColumn* objects. It provides methods that make accessing columns easier than if they were stored in a `list` or `dict`. `Columns` has a similar API to a `dict` (it actually uses a `OrderedDict` internally).

At the moment you'll only come across this class when you access a `Table.columns` property.

Parameters **table** (`Table`) – the table containing the columns

__contains__(*item*)
Check if a column is contained within a *BoundColumns* object.

item can either be a *BoundColumn* object, or the name of a column.

__getitem__(*index*)
Retrieve a specific *BoundColumn* object.

index can either be 0-indexed or the name of a column

```
columns['speed']  # returns a bound column with name 'speed'
columns[0]        # returns the first column
```

__init__(*table*, *base_columns*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()
Convenience API, alias of *itervisible*.

__len__()
Return how many *BoundColumn* objects are contained (and visible).

__weakref__
list of weak references to the object (if defined)

hide(*name*)
Hide a column.

Parameters **name** (*str*) – name of the column

iterall()
Return an iterator that exposes all *BoundColumn* objects, regardless of visibility or sortability.

iteritems()
Return an iterator of `(name, column)` pairs (where `column` is a *BoundColumn*).

This method is the mechanism for retrieving columns that takes into consideration all of the ordering and filtering modifiers that a table supports (e.g. `exclude` and `sequence`).

iterorderable()
Same as `BoundColumns.all` but only returns orderable columns.

This is useful in templates, where iterating over the full set and checking `{% if column.ordarable %}` can be problematic in conjunction with e.g. `{{ forloop.last }}` (the last column might not be the actual last that is rendered).

itervisible()
Same as *iterorderable* but only returns visible *BoundColumn* objects.

This is geared towards table rendering.

show(*name*)
Show a column otherwise hidden.

Parameters **name** (*str*) – name of the column

BoundColumn

class django_tables2.columns.**BoundColumn**(*table*, *column*, *name*)
A *run-time* version of *Column*. The difference between *BoundColumn* and *Column*, is that *BoundColumn* objects include the relationship between a *Column* and a Table. In practice, this means that a *BoundColumn* knows the *"variable name"* given to the *Column* when it was declared on the Table.

Parameters

- **table** (Table) – The table in which this column exists
- **column** (*Column*) – The type of column
- **name** (*str*) – The variable name of the column used when defining the Table. In this example the name is age:

```
class SimpleTable(tables.Table):
    age = tables.Column()
```

__init__(*table*, *column*, *name*)
Initialize self. See help(type(self)) for accurate signature.

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

_get_cell_class(*attrs*)
Return a set of the classes from the class key in attrs.

property accessor
Returns the string used to access data for this column out of the data source.

property attrs
Proxy to Column.attrs but injects some values of our own.

A th, td and tf are guaranteed to be defined (irrespective of what is actually defined in the column attrs. This makes writing templates easier. tf is not actually a HTML tag, but this key name will be used for attributes for column's footer, if the column has one.

property default
Returns the default value for this column.

get_td_class(*td_attrs*)
Returns the HTML class attribute for a data cell in this column

get_th_class(*th_attrs*)
Returns the HTML class attribute for a header cell in this column

property header
The value that should be used in the header cell for this column.

property localize
Returns `True`, `False` or `None` as described in `Column.localize`

property order_by
Returns an *`OrderByTuple`* of appropriately prefixed data source keys used to sort this column.

See *`order_by_alias`* for details.

property order_by_alias
Returns an `OrderBy` describing the current state of ordering for this column.

The following attempts to explain the difference between *`order_by`* and *`order_by_alias`*.

`order_by_alias` returns and *`OrderBy`* instance that's based on the *name* of the column, rather than the keys used to order the table data. Understanding the difference is essential.

Having an alias *and* a keys version is necessary because an N-tuple (of data source keys) can be used by the column to order the data, and it is ambiguous when mapping from N-tuple to column (since multiple columns could use the same N-tuple).

The solution is to use order by *aliases* (which are really just prefixed column names) that describe the ordering *state* of the column, rather than the specific keys in the data source should be ordered.

e.g.:

```
>>> class SimpleTable(tables.Table):
...     name = tables.Column(order_by=("firstname", "last_name"))
...
>>> table = SimpleTable([], order_by=('-name', ))
>>> table.columns["name"].order_by_alias
"-name"
>>> table.columns["name"].order_by
("-first_name", "-last_name")
```

The `OrderBy` returned has been patched to include an extra attribute `next`, which returns a version of the alias that would be transitioned to if the user toggles sorting on this column, for example:

```
not sorted -> ascending
ascending  -> descending
descending -> ascending
```

This is useful otherwise in templates you'd need something like:

```
{% if column.is_ordered %}
{% querystring table.prefixed_order_by_field=column.order_by_alias.opposite %}
{% else %}
{% querystring table.prefixed_order_by_field=column.order_by_alias %}
{% endif %}
```

property orderable
Return a `bool` depending on whether this column supports ordering.

property verbose_name
Return the verbose name for this column.

In order of preference, this will return:

1) The column's explicitly defined *`verbose_name`*
2) The model's *`verbose_name`* with the first letter capitalized (if applicable)
3) Fall back to the column name, with first letter capitalized.

Any *verbose_name* that was not passed explicitly in the column definition is returned with the first character capitalized in keeping with the Django convention of *verbose_name* being defined in lowercase and uppercased as needed by the application.

If the table is using `QuerySet` data, then use the corresponding model field's `verbose_name`. If it is traversing a relationship, then get the last field in the accessor (i.e. stop when the relationship turns from ORM relationships to object attributes [e.g. person.upper should stop at person]).

property visible
Returns a `bool` depending on whether this column is visible.

BoundRows

class django_tables2.rows.**BoundRows**(*data*, *table*, *pinned_data=None*)
Container for spawning *BoundRow* objects.

Parameters

- **data** – iterable of records
- **table** – the `Table` in which the rows exist
- **pinned_data** – dictionary with iterable of records for top and/or bottom pinned rows.

Example

```
>>> pinned_data = {
...     'top': iterable,       # or None value
...     'bottom': iterable,    # or None value
... }
```

This is used for `rows`.

__getitem__(*key*)
Slicing returns a new *BoundRows* instance, indexing returns a single *BoundRow* instance.

__init__(*data*, *table*, *pinned_data=None*)
Initialize self. See help(type(self)) for accurate signature.

__weakref__
list of weak references to the object (if defined)

generator_pinned_row(*data*)
Top and bottom pinned rows generator.

Parameters **data** – Iterable data for all records for top or bottom pinned rows.

Yields *BoundPinnedRow* – Top or bottom *BoundPinnedRow* object for single pinned record.

BoundRow

class django_tables2.rows.**BoundRow**(*record*, *table*)
Represents a *specific* row in a table.

BoundRow objects are a container that make it easy to access the final 'rendered' values for cells in a row. You can simply iterate over a *BoundRow* object and it will take care to return values rendered using the correct method (e.g. *Table.render_foo methods*)

To access the rendered value of each cell in a row, just iterate over it:

```
>>> import django_tables2 as tables
>>> class SimpleTable(tables.Table):
...     a = tables.Column()
...     b = tables.CheckBoxColumn(attrs={'name': 'my_chkbox'})
...
>>> table = SimpleTable([{'a': 1, 'b': 2}])
>>> row = table.rows[0]  # we only have one row, so let's use it
>>> for cell in row:
...     print(cell)
...
1
<input type="checkbox" name="my_chkbox" value="2" />
```

Alternatively you can use row.cells[0] to retrieve a specific cell:

```
>>> row.cells[0]
1
>>> row.cells[1]
'<input type="checkbox" name="my_chkbox" value="2" />'
>>> row.cells[2]
...
IndexError: list index out of range
```

Finally you can also use the column names to retrieve a specific cell:

```
>>> row.cells.a
1
>>> row.cells.b
'<input type="checkbox" name="my_chkbox" value="2" />'
>>> row.cells.c
...
KeyError: "Column with name 'c' does not exist; choices are: ['a', 'b']"
```

If you have the column name in a variable, you can also treat the `cells` property like a `dict`:

```
>>> key = 'a'
>>> row.cells[key]
1
```

Parameters

- **table** – The `Table` in which this row exists.
- **record** – a single record from the *table data* that is used to populate the row. A record could be a `Model` object, a `dict`, or something else.

__contains__(*item*)
Check by both row object and column name.

__init__(*record*, *table*)
Initialize self. See help(type(self)) for accurate signature.

__iter__()
Iterate over the rendered values for cells in the row.

Under the hood this method just makes a call to `BoundRow.__getitem__` for each cell.

__weakref__
list of weak references to the object (if defined)

_call_render(*bound_column*, *value=None*)
Call the column's render method with appropriate kwargs

_call_value(*bound_column*, *value=None*)
Call the column's value method with appropriate kwargs

_optional_cell_arguments(*bound_column*, *value*)
Defines the arguments that will optionally be passed while calling the cell's rendering or value getter if that function has one of these as a keyword argument.

property attrs
Return the attributes for a certain row.

get_cell(*name*)
Returns the final rendered html for a cell in the row, given the name of a column.

get_cell_value(*name*)
Returns the final rendered value (excluding any html) for a cell in the row, given the name of a column.

get_even_odd_css_class()
Return css class, alternating for odd and even records.

Returns `even` for even records, `odd` otherwise.

Return type string

items()
Returns iterator yielding `(bound_column, cell)` pairs.

cell is `row[name]` – the rendered unicode value that should be `rendered within ``<td>`.

property record
The data record from the data source which is used to populate this row with data.

property table
The associated `Table` object.

BoundPinnedRow

class django_tables2.rows.**BoundPinnedRow**(*record*, *table*)
Represents a *pinned* row in a table.

property attrs
Return the attributes for a certain pinned row. Add CSS classes `pinned-row` and `odd` or `even` to `class` attribute.

Returns Attributes for pinned rows.

Return type AttributeDict

TableData

utils

class django_tables2.utils.**Sequence**
Represents a column sequence, e.g. `('first_name', '...', 'last_name')`

This is used to represent `Table.Meta.sequence` or the `Table` constructors's *sequence* keyword argument.

The sequence must be a list of column names and is used to specify the order of the columns on a table. Optionally a '...' item can be inserted, which is treated as a *catch-all* for column names that are not explicitly specified.

__weakref__
list of weak references to the object (if defined)

expand(*columns*)
Expands the `'...'` item in the sequence into the appropriate column names that should be placed there.

Parameters **columns** (*list*) – list of column names.

Returns The current instance.

Raises **ValueError** –

class django_tables2.utils.**OrderBy**
A single item in an *OrderByTuple* object.

This class is essentially just a `str` with some extra properties.

static __new__(*cls*, *value*)
Create and return a new object. See help(type) for accurate signature.

__weakref__
list of weak references to the object (if defined)

property bare

Returns the bare form.

Return type *OrderBy*

The *bare form* is the non-prefixed form. Typically the bare form is just the ascending form.

Example: `age` is the bare form of `-age`

for_queryset()
Returns the current instance usable in Django QuerySet's order_by arguments.

property is_ascending
Returns `True` if this object induces *ascending* ordering.

property is_descending
Returns `True` if this object induces *descending* ordering.

property opposite
Provides the opposite of the current sorting direction.

Returns object with an opposite sort influence.

Return type *OrderBy*

Example:

```
>>> order_by = OrderBy('name')
>>> order_by.opposite
'-name'
```

class django_tables2.utils.**OrderByTuple**
Stores ordering as (as *OrderBy* objects).

The `order_by` property is always converted to an *OrderByTuple* object. This class is essentially just a `tuple` with some useful extras.

Example:

```
>>> x = OrderByTuple(('name', '-age'))
>>> x['age']
'-age'
>>> x['age'].is_descending
True
>>> x['age'].opposite
'age'
```

__contains__(*name*)

Determine if a column has an influence on ordering.

Example:

```
>>> x = OrderByTuple(('name', ))
>>> 'name' in  x
True
>>> '-name' in x
True
```

Parameters **name** (*str*) – The name of a column. (optionally prefixed)

Returns `True` if the column with `name` influences the ordering.

Return type bool

__getitem__(*index*)

Allows an *OrderBy* object to be extracted via named or integer based indexing.

When using named based indexing, it's fine to used a prefixed named:

```
>>> x = OrderByTuple(('name', '-age'))
>>> x[0]
'name'
>>> x['age']
'-age'
>>> x['-age']
'-age'
```

Parameters **index** (*int*) – Index to query the ordering for.

Returns for the ordering at the index.

Return type *OrderBy*

static __new__(*cls*, *iterable*)

Create and return a new object. See help(type) for accurate signature.

__str__()

Return str(self).

get(*key*, *fallback*)

Identical to *__getitem__*, but supports fallback value.

property opposite

Return version with each *OrderBy* prefix toggled:

```
>>> order_by = OrderByTuple(('name', '-age'))
>>> order_by.opposite
('-name', 'age')
```

class django_tables2.utils.**Accessor**

A string describing a path from one object to another via attribute/index accesses. For convenience, the class has an alias `A` to allow for more concise code.

Relations are separated by a `__` character.

To support list-of-dicts from `QuerySet.values()`, if the context is a dictionary, and the accessor is a key in the dictionary, it is returned right away.

static __new__(*cls*, *value*)

Create and return a new object. See help(type) for accurate signature.

__weakref__

list of weak references to the object (if defined)

get_field(*model*)

Return the django model field for model in context, following relations.

penultimate(*context*, *quiet=True*)

Split the accessor on the right-most separator ('__'), return a tuple with:

- the resolved left part.
- the remainder

Example:

```
>>> Accessor("a__b__c").penultimate({"a": {"a": 1, "b": {"c": 2, "d": 4}}})
({"c": 2, "d": 4}, "c")
```

resolve(*context*, *safe=True*, *quiet=False*)

Return an object described by the accessor by traversing the attributes of *context*.

Lookups are attempted in the following order:

- dictionary (e.g. `obj[related]`)
- attribute (e.g. `obj.related`)
- list-index lookup (e.g. `obj[int(related)]`)

Callable objects are called, and their result is used, before proceeding with the resolving.

Example:

```
>>> x = Accessor("__len__")
>>> x.resolve("brad")
4
>>> x = Accessor("0__upper")
>>> x.resolve("brad")
"B"
```

If the context is a dictionary and the accessor-value is a key in it, the value for that key is immediately returned:

```
>>> x = Accessor("user__first_name")
>>> x.resolve({"user__first_name": "brad"})
"brad"
```

Parameters

- **context** – The root/first object to traverse.

- **safe** (*bool*) – Don't call anything with `alters_data = True`
- **quiet** (*bool*) – Smother all exceptions and instead return `None`

Returns target object

Raises

- **TypeError`, AttributeError, KeyError, ValueError** –
- **(unless quiet == True)** –

class `django_tables2.utils.`**AttributeDict**

A wrapper around `collections.OrderedDict` that knows how to render itself as HTML style tag attributes.

Any key with `value is None` will be skipped.

The returned string is marked safe, so it can be used safely in a template. See `as_html` for a usage example.

as_html()

Render to HTML tag attributes.

Example:

```
>>> from django_tables2.utils import AttributeDict
>>> attrs = AttributeDict({'class': 'mytable', 'id': 'someid'})
>>> attrs.as_html()
'class="mytable" id="someid"'
```

returns: `SafeUnicode` object

`django_tables2.utils.`**signature**(*fn*)

Returns

Returns a (arguments, kwarg_name)-tuple:

- the arguments (positional or keyword)
- the name of the ** kwarg catch all.

Return type tuple

The self-argument for methods is always removed.

`django_tables2.utils.`**call_with_appropriate**(*fn*, *kwargs*)

Calls the function `fn` with the keyword arguments from `kwargs` it expects

If the kwargs argument is defined, pass all arguments, else provide exactly the arguments wanted.

If one of the arguments of `fn` are not contained in kwargs, `fn` will not be called and `None` will be returned.

`django_tables2.utils.`**computed_values**(*d*, *kwargs=None*)

Returns a new `dict` that has callable values replaced with the return values.

Example:

```
>>> compute_values({'foo': lambda: 'bar'})
{'foo': 'bar'}
```

Arbitrarily deep structures are supported. The logic is as follows:

1. If the value is callable, call it and make that the new value.
2. If the value is an instance of dict, use ComputableDict to compute its keys.

Example:

```
>>> def parents():
...     return {
...         'father': lambda: 'Foo',
...         'mother': 'Bar'
...      }
...
>>> a = {
...     'name': 'Brad',
...     'parents': parents
... }
...
>>> computed_values(a)
{'name': 'Brad', 'parents': {'father': 'Foo', 'mother': 'Bar'}}
```

Parameters

- **d** (*dict*) – The original dictionary.
- **kwargs** – any extra keyword arguments will be passed to the callables, if the callable takes an argument with such a name.

Returns with callable values replaced.

Return type dict

1.19 FAQ

Some frequently requested questions/examples. All examples assume you import django-tables2 like this:

```
import django_tables2 as tables
```

1.19.1 How should I fix error messages about the request context processor?

The error message looks something like this:

```
Tag {% querystring %} requires django.template.context_processors.request to be
in the template configuration in settings.TEMPLATES[]OPTIONS.context_processors)
in order for the included template tags to function correctly.
```

which should be pretty clear, but here is an example template configuration anyway:

```
TEMPLATES = [
    {
        "BACKEND": "django.template.backends.django.DjangoTemplates",
        "DIRS": ["templates"],
        "APP_DIRS": True,
        "OPTIONS": {
            "context_processors": [
                "django.contrib.auth.context_processors.auth",
                "django.template.context_processors.request",
                "django.template.context_processors.static",
            ],
```

(continues on next page)

(continued from previous page)

```
        }
    }
]
```

1.19.2 How to create a row counter?

You can use `itertools.counter` to add row count to a table. Note that in a paginated table, every page's counter will start at zero:

```
class CountryTable(tables.Table):
    counter = tables.TemplateColumn("{{ row_counter }}")
```

1.19.3 How to add a footer containing a column total?

Using the `footer`-argument to *Column*:

```
class CountryTable(tables.Table):
    population = tables.Column(
        footer=lambda table: sum(x["population"] for x in table.data)
    )
```

Or by creating a custom column:

```
class SummingColumn(tables.Column):
    def render_footer(self, bound_column, table):
        return sum(bound_column.accessor.resolve(row) for row in table.data)

class Table(tables.Table):
    name = tables.Column(footer="Total:")
    population = SummingColumn()
```

Documentation: *Adding column footers*

Note: Your table template must include a block rendering the table footer!

1.19.4 Can I use inheritance to build Tables that share features?

Yes, like this:

```
class CountryTable(tables.Table):
    name = tables.Column()
    language = tables.Column()
```

A `CountryTable` will show columns `name` and `language`:

```
class TouristCountryTable(CountryTable):
    tourist_info = tables.Column()
```

A `TouristCountryTable` will show columns `name`, `language` and `tourist_info`.

Overwriting a `Column` attribute from the base class with anything that is not a `Column` will result in removing that Column from the `Table`. For example:

```
class SimpleCountryTable(CountryTable):
    language = None
```

A `SimpleCountryTable` will only show column `name`.

1.19.5 How can I use with Jinja2 template?

In Jinja2 templates, the `{% render_table %}` tag is not available, but you can still use *django-tables2* like this:

```
{{ table.as_html(request) }}
```

where `request` need to be passed from view, or from *context processors* (which is supported by django-jinja).

1.20 Upgrading and change log

Recent versions of django-tables2 have a corresponding git tag for each version released to pypi.

1.21 Glossary

accessor Refers to an `Accessor` object

column name The name given to a column. In the follow example, the *column name* is `age`.

```
class SimpleTable(tables.Table):
    age = tables.Column()
```

empty value An empty value is synonymous with "no value". Columns have an `empty_values` attribute that contains values that are considered empty. It's a way to declare which values from the database correspond to *null/blank/missing* etc.

order by alias A prefixed column name that describes how a column should impact the order of data within the table. This allows the implementation of how a column affects ordering to be abstracted, which is useful (e.g. in query strings).

```
class ExampleTable(tables.Table):
    name = tables.Column(order_by=("first_name", "last_name"))
```

In this example `-name` and `name` are valid order by aliases. In a query string you might then have `?order=-name`.

table The traditional concept of a table. i.e. a grid of rows and columns containing data.

view A Django view.

record A single Python object used as the data for a single row.

render The act of serializing a `Table` into HTML.

template A Django template.

table data An iterable of *records* that `Table` uses to populate its rows.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

O

R

S

T

U

V