Django-Chartit Documentation

Release 0.2.9

Praveen Gollakota

Apr 03, 2018

Contents

Django Chartit is a Django app that can be used to easily create charts from the data in your database. The charts are rendered using `Highcharts` and `jQuery` JavaScript libraries. Data in your database can be plotted as simple line charts, column charts, area charts, scatter plots, and many more chart types. Data can also be plotted as Pivot Charts where the data is grouped and/or pivoted by specific column(s).

CHAPTER 1

Changelog

- **master**
  - Update demo with an example of how to pass `legendIndex` as an option to a data serie. Closes #48.
  - Update demo with an example of how to change the label of any term instead of using the default one. Closes #46.
- **0.2.9 (January 17, 2017)**
  - Enable pylint during testing but don't block Travis-CI on failures. Closes #42.
  - Handle unicode data in pie and scatter plot charts under Python 2.7. PR#47.
- **0.2.8 (December 4, 2016)**
  - `PivotChart` and `PivotDataPool` **will be deprecated soon**. Both are marked with deprecation warnings. There is a lot of duplication and special handling between those classes and the `Chart` and `DataPool` classes which make it harder to expand the feature set for django-chartit. The next release will focus on consolidating all the functionality into `Chart` and `DataPool` so that users will still be able to draw pivot charts. You will have to construct your pivot charts manually though!
  - `DataPool` terms now supports model properties. Fixes #35. Model properties are **not** supported for `PivotDataPool`! **WARNING: when using model properties chartit can't make use of ''QuerySet.values()'' internally. This means results will not be groupped by the values of the fields you supplied. This may lead to unexpected query results/charts!**
  - `DataPool` now supports `RawQuerySet` as data source. Fixes #44. `RawQuerySet` is **not** supported for `PivotDataPool`! **WARNING: when using ''RawQuerySet'' don't use double underscores in field names because these are interpreted internally by chartit and will cause exceptions. For example don't do this ''SELECT AVG(rating) as rating__avg'' instead write it as ''SELECT AVG(rating) as rating_avg''!**
  - README now tells how to execute `demoproject/`
- **0.2.7 (September 14, 2016)**
  - Don't use `super(self.__class__)` b/c that breaks chart class inheritance. Fixes #41

- **0.2.6 (August 16, 2016)**
  - Merge `chartit_tests/` with `demoproject/`
  - Load test DB with real data to use during testing
  - Add more tests
  - Update the path to demoproject.settings when building docs. Fixes a problem which caused some API docs to be empty
  - Fix ValueError: not enough values to unpack (expected 2, got 0) with PivotChart when the QuerySet returns empty data
  - Dropped requirement on `simplejson`
  - Properly handle unicode data in Pivot charts. Fixes #5
  - Demo project updated with Chart and PivotChart examples of rendering DateField values on the X axis
  - Allow charting of `extra()` or `annotate()` fields. Fixes #8 and #12
  - Refactor `RecursiveDefaultDict` to allow chart objects to be serialized to/from cache. Fixes #10
  - Add information about supported 3rd party JavaScript versions. Fixes #14
- **0.2.5 (August 3, 2016)**
  - Workaround Python 3 vs. Python 2 list sort issue which breaks charts with multiple data sources displayed on the same axis!
  - Make demoproject/ compatible with Django 1.10
- **0.2.4 (August 2, 2016)**
  - Fix for `get_all_field_names()` and `get_field_by_name()` removal in Django 1.10. Fixes #39
  - Updated for django.db.sql.query.Query.aggregates removal
- **0.2.3 (July 30, 2016)**
  - New to_json() method for charts. Useful for creating Highcharts in AJAX
  - Merged with *django-chartit2* fork by Grant McConnaughey which adds Python 3 and latest Django 1.8.x and 1.9.x support
  - Allow dictionary fields in conjunction with lambda fields. Closes #26
  - Documentation improvements
  - Lots of code cleanups and style improvements
- **0.2.2 as django-chartit2 (January 28, 2016)**
  - Fixed another issue that prevented installation via PyPI
- **0.2.0 as django-chartit2 (January 20, 2016):**
  - Fixed issue that could prevent installation via PyPI
- **0.1 (November 5, 2011)**
  - Initial release of django-chartit

CHAPTER 2

Features

- Plot charts from models.
- Plot data from multiple models on the same axis on a chart.
- Plot pivot charts from models. Data can be pivoted by across multiple columns.
- Legend pivot charts by multiple columns.
- Combine data from multiple models to plot on same pivot charts.
- Plot a pareto chart, paretoed by a specific column.
- Plot only a top few items per category in a pivot chart.
- Python 3 compatibility
- Django 1.8 and 1.9 compatibility
- Documentation to ReadTheDocs
- Automated testing via Travis CI
- Test coverage tracking via Coveralls

CHAPTER 3

Installation

You can install Django-Chartit from PyPI. Just do

```
$ pip install django_chartit
```

Then, add *chartit* to *INSTALLED_APPS* in "settings.py".

You also need supporting JavaScript libraries. See the *Required JavaScript Libraries* section for more details.

CHAPTER 4

How to Use

Plotting a chart or pivot chart on a webpage involves the following steps.

1. Create a `DataPool` or `PivotDataPool` object that specifies what data you need to retrieve and from where.
2. Create a `Chart` or `PivotChart` object to plot the data in the `DataPool` or `PivotDataPool` respectively.
3. Return the `Chart`/`PivotChart` object from a django `view` function to the django template.
4. Use the `load_charts` template tag to load the charts to HTML tags with specific *ids*.

It is easier to explain the steps above with examples. So read on.

CHAPTER 5

How to Create Charts

Here is a short example of how to create a line chart. Let's say we have a simple model with 3 fields - one for month and two for temperatures of Boston and Houston.

```
class MonthlyWeatherByCity(models.Model):
    month = models.IntegerField()
    boston_temp = models.DecimalField(max_digits=5, decimal_places=1)
    houston_temp = models.DecimalField(max_digits=5, decimal_places=1)
```

And let's say we want to create a simple line chart of month on the x-axis and the temperatures of the two cities on the y-axis.

```
from chartit import DataPool, Chart

def weather_chart_view(request):
    #Step 1: Create a DataPool with the data we want to retrieve.
    weatherdata = \
        DataPool(
           series=
            [{'options': {
               'source': MonthlyWeatherByCity.objects.all()},
              'terms': [
                'month',
                'houston_temp',
                'boston_temp']}
             ])

    #Step 2: Create the Chart object
    cht = Chart(
            datasource = weatherdata,
            series_options =
              [{'options':{
                  'type': 'line',
                  'stacking': False},
                'terms':{
                  'month': [
```

```
                    'boston_temp',
                    'houston_temp']
                }}],
            chart_options =
              {'title': {
                   'text': 'Weather Data of Boston and Houston'},
               'xAxis': {
                    'title': {
                       'text': 'Month number'}}})

    #Step 3: Send the chart object to the template.
    return render_to_response({'weatherchart': cht})
```

And you can use the `load_charts` filter in the django template to render the chart.

```
<head>
    <!-- code to include the highcharts and jQuery libraries goes here -->
    <!-- load_charts filter takes a comma-separated list of id's where -->
    <!-- the charts need to be rendered to                             -->
    {% load chartit %}
    {{ weatherchart|load_charts:"container" }}
</head>
<body>
    <div id='container'> Chart will be rendered here </div>
</body>
```

CHAPTER 6

How to Create Pivot Charts

Here is an example of how to create a pivot chart. Let's say we have the following model.

```
class DailyWeather(models.Model):
    month = models.IntegerField()
    day = models.IntegerField()
    temperature = models.DecimalField(max_digits=5, decimal_places=1)
    rainfall = models.DecimalField(max_digits=5, decimal_places=1)
    city = models.CharField(max_length=50)
    state = models.CharField(max_length=2)
```

We want to plot a pivot chart of month (along the x-axis) versus the average rainfall (along the y-axis) of the top 3 cities with highest average rainfall in each month.

```
from django.db.models import Avg
from chartit import PivotDataPool, PivotChart

def rainfall_pivot_chart_view(request):
    # Step 1: Create a PivotDataPool with the data we want to retrieve.
    rainpivotdata = PivotDataPool(
        series=[{
            'options': {
                'source': DailyWeather.objects.all(),
                'categories': ['month'],
                'legend_by': 'city',
                'top_n_per_cat': 3,
            },
            'terms': {
                'avg_rain': Avg('rainfall'),
            }
        }]
    )

    # Step 2: Create the PivotChart object
    rainpivcht = PivotChart(
        datasource=rainpivotdata,
```

```
        series_options=[{
            'options': {
                'type': 'column',
                'stacking': True
            },
            'terms': ['avg_rain']
        }],
        chart_options={
            'title': {
                'text': 'Rain by Month in top 3 cities'
            },
            'xAxis': {
                'title': {
                    'text': 'Month'
                }
            }
        }
    )

    # Step 3: Send the PivotChart object to the template.
    return render_to_response({'rainpivchart': rainpivcht})
```

And you can use the `load_charts` filter in the django template to render the chart.

```
<head>
    <!-- code to include the highcharts and jQuery libraries goes here -->
    <!-- load_charts filter takes a comma-separated list of id's where -->
    <!-- the charts need to be rendered to                             -->
    {% load chartit %}
    {{ rainpivchart|load_charts:"container" }}
</head>
<body>
    <div id='container'> Chart will be rendered here </div>
</body>
```

CHAPTER 7

Rendering multiple charts

It is possible to render multiple charts in the same template. The first argument to `load_charts` is the Chart object or a list of Chart objects, and the second is a comma separated list of HTML IDs where the charts will be rendered.

When calling Django's `render` you have to pass all you charts as a list:

```
return render(request, 'index.html',
              {
                 'chart_list' : [chart_1, chart_2],
              }
        )
```

Then in your template you have to use the proper syntax:

```
<head>
    {% load chartit %}
    {{ chart_list|load_charts:"chart_1,chart_2" }}
</head>
<body>
    <div id="chart_1">First chart will be rendered here</div>
    <div id="chart_2">Second chart will be rendered here</div>
</body>
```

CHAPTER 8

Demo

The above examples are just a brief taste of what you can do with Django-Chartit. For more examples and to look at the charts in actions, check out the `demoproject/` directory. To execute the demo run the commands

```
cd demoproject/
PYTHONPATH=../ python ./manage.py migrate
PYTHONPATH=../ python ./manage.py runserver
```

CHAPTER 9

Documentation

Full documentation is available here .

CHAPTER 10

Required JavaScript Libraries

The following JavaScript Libraries are required for using Django-Chartit.

- jQuery - versions 1.6.4 and 1.7 are known to work well with django-chartit.
- Highcharts - versions 2.1.7 and 2.2.0 are known to work well with django-chartit.

Note: While `Django-Chartit` itself is licensed under the BSD license, `Highcharts` is licensed under the Highcharts license and `jQuery` is licensed under both MIT License and GNU General Public License (GPL) Version 2. It is your own responsibility to abide by respective licenses when downloading and using the supporting JavaScript libraries.

CHAPTER 11

Quick API Reference

11.1 API Reference

11.1.1 How to retrieve data

DataPool

`DataPool.__init__`(*series*)

Create a DataPool object as specified by the `series`.

Arguments

- **series** *(list of dict)* - specifies the what data to retrieve and where to retrieve it from. It is of the form

```
[{'options': {
   'source': a django model, Manager or QuerySet,
   },
 'terms': [
   'a_valid_field_name', ... ,
   {'any_name': 'a_valid_field_name', ... },
   ]
},
...
]
```

Where

  - **options** (**required**) - a `dict`. Any of the series options for the Highcharts `options` object are valid.
  - **terms** - is a list. Each element in `terms` is either
    1. a `str` - needs to be a valid model field for the corresponding `source`, or
    2. a `dict` - need to be of the form `{'any_name': 'a_valid_field_name', ...}`.

To retrieve data from multiple models or QuerySets, just add more dictionaries with the corresponding `options` and terms.

Raises

- **APIInputError** - sif the `series` argument has any invalid parameters.

> **Warning:** All elements in `terms` **must be unique** across all the dictionaries in the `series` list. If there are two terms with same `name`, the latter one is going to overwrite the one before it.

For example, the following is **wrong**:

```
[{'options': {
    'source': SomeModel},
  'terms':[
    'foo',
    'bar']},
 {'options': {
    'source': OtherModel},
  'terms':[
    'foo']}]
```

In this case, the term `foo` from `OtherModel` is going to **overwrite** `foo` from `SomeModel`.

Here is the **right** way of retrieving data from two different models both of which have the same field name.

```
[{'options': {
   'source': SomeModel},
  'terms':[
    'foo',
    'bar']},
 {'options': {
   'source': OtherModel},
  'terms':[
    {'foo_2': 'foo'}]}]
```

PivotDataPool

`PivotDataPool.__init__`(*series*, *top_n_term=None*, *top_n=None*, *pareto_term=None*, *sortf_mapf_mts=None*)

Creates a PivotDataPool object.

Arguments

- **series** (**required**) - a list of dicts that specifies the what data to retrieve, where to retrieve it from and how to pivot the data. It is of the form

```
[{'options': {
    'source': django Model, Manager or QuerySet ,
    'categories': ['a_valid_field', ...],
    'legend_by': ['a_valid_field', ...] (optional),
    'top_n_per_cat': a number (optional),
  },
  'terms': {
    'any_name_here': django Aggregate,
```

```
    'some_other_name':{
      'func': django Aggregate,
      #any options to override
      ...
    },
  ...
  }
 },
 ... #repeat dicts with 'options' & 'terms'
]
```

Where

- **options** - is a dict that specifies the common options for all the terms.
  - **source** (**required**) - is either a `Model`, `Manager` or a `QuerySet`.
  - **categories** (**required**) - is a list of model fields by which the data needs to be pivoted by. If there is only a single item, `categories` can just be a string instead of a list with single element.

    For example if you have a model with `country`, `state`, `county`, `city`, `date`, `rainfall`, `temperature` and you want to pivot the data by `country` and `state`, then `categories = ['country', 'state']`.

    Note: Order of elements in the `categories` list matters!

    `categories = ['country', 'state']` groups your data first by `country` and then by `state` when running the SQL query. This obviously is not the same as grouping by `state` first and then by `country`.
  - **legend_by** (*optional*) - is a list of model fields by which the data needs to be legended by. For example, in the above case, if you want to legend by `county` and `city`, then `legend_by = ['county', 'city']`

    Note: Order of elements in the `legend_by` list matters!

    See the note in `categories` above.
  - **top_n_per_cat** (*optional*) - The number of top items that the legended entries need to be limited to in each category. For example, in the above case, if you wanted only the top 3 `county/cities` with highest rainfall for each of the `country/state`, then `top_n_per_cat = 3`.
- **terms** - is a `dict`. The keys can be any strings (but helps if they are meaningful aliases for the field). The values can either be
  - a django `Aggregate` : of a valid field in corresponding model. For example, `Avg('temperature')`, `Sum('price')`, etc. or
  - a `dict`: In this case the `func` must specify relevant django aggregate to retrieve. For example `'func': Avg('price')`. The dict can also have any additional entries from the options dict. Any entries here will override the entries in the `options` dict.
- **top_n_term** (*optional*) - a string. Must be one of the keys in the corresponding `terms` in the `series` argument.
- **top_n** (*optional*) - an integer. The number of items for the corresponding `top_n_term` that need to be retained.

If `top_n_term` and `top_n` are present, only the `top_n` numberof items are going to displayed in the pivot chart. For example, if you want to plot only the top 5 states with highest average rainfall, you can do something like this.

```
PivotDataPool(
  series = [
     {'options': {
        'source': RainfallData.objects.all(),
        'categories': 'state'},
      'terms': {
        'avg_rain': Avg('rainfall')}}],
  top_n_term = 'avg_rain',
  top_n = 5)
```

Note that the `top_n_term` is `'avg_rain'` and **not** `state`; because we want to limit by the average rainfall.

- **pareto_term** (*optional*) - the term with respect to which the pivot chart needs to be paretoed by.

  For example, if you want to plot the average rainfall on the y-axis w.r.t the state on the x-axis and want to pareto by the average rainfall, you can do something like this.

```
PivotDataPool(
  series = [
     {'options': {
        'source': RainfallData.objects.all(),
        'categories': 'state'},
      'terms': {
        'avg_rain': Avg('rainfall')}}],
  pareto_term = 'avg_rain')
```

- **sortf_mapf_mts** (*optional*) - a `tuple` with three elements of the form `(sortf, mapf, mts)` where
  - **sortf** - is a function (or a callable) that is used as a *key* when sorting the category values.

    For example, if `categories = 'month_num'` and if the months need to be sorted in reverse order, then `sortf` can be

```
sortf = lambda *x: (-1*x[0],)
```

    Note: `sortf` is passed the category values as tuples and must return tuples!

    If `categories` is `['city', 'state']` and if the category values returned need to be sorted with state first and then city, then `sortf` can be

```
sortf = lambda *x: (x[1], x[0])
```

    The above `sortf` is passed tuples like `('San Francisco', 'CA')`, `('New York', 'NY')`, `...` and it returns tuples like `('CA', 'San Francisco')`, `('NY', 'New York')`, `...` which when used as keys to sort the category values will obviously first sort by state and then by city.

  - **mapf** - is a function (or a callable) that defines how the category values need to be mapped.

    For example, let's say `categories` is `'month_num'` and that the category values that are retrieved from your database are `1`, `2`, `3`, etc. If you want month *names* as the category values instead of month numbers, you can define a `mapf` to transform the month numbers to month names like so

```
def month_name(*t):
    names ={1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr',
            5: 'May', 6: 'Jun', 7: 'Jul', 8: 'Aug',
            9: 'Sep', 10: 'Oct', 11: 'Nov', 12: 'Dec'}
    month_num = t[0]
    return (names[month_num], )

mapf = month_name
```

Note: `mapf` like `sortf` is passed the category values as tuples and must return tuples.

– **mts** - *map then sort* ; a `bool`. If `True`, the category values are mapped first and then sorted, and if `False` category values are sorted first and then mapped.

In the above example of month names, we `mts` must be `False` because the months must first be sorted based on their number and then mapped to their names. If `mts` is `True`, the month numbers would be transformed to the month names, and then sorted, which would yield an order like `Apr`, `Aug`, `Dec`, etc. (not what we want).

Raises

- **APIInputError** - if the `series` argument has any invalid parameters.

Here is a full example of a `series` term that retrieves the average temperature of the top 3 cities in each country/state and the average rainfall of the top 2 cities in each country/state.

```
[{'options': {
    'source': Weather.objects.all(),
    'categories': ['country', 'state'],
    'legend_by': 'city',
    'top_n_per_cat': 3},
  'terms': {
    'avg_temp': Avg('temperature'),
    'avg_rain': {
      'func': Avg('rainfall'),
      'top_n_per_cat': 2}}}]
```

The `'top_n_per_cat': 2` term in `avg_rain` dict overrides `'top_n_per_cat': 5` from the comon options dict. Effectively, the above `series` retrieves the *top 2* `cities` with highest `avg_rain` in each `country/state` and *top 3* `cities` with highest `avg_temp` in each `country/state`.

A single `PivotDataPool` can hold data from multiple Models. If there are more models or QuerySets to retrieve the data from, just add more dicts to the series list with different `source` values.

Warning: The `keys` for the `terms` must be **unique across all the dictionaries** in the `series` list! If there are multiple terms with same `key`, the latter ones will just overwrite the previous ones.

For instance, the following example is **wrong**.

```
[{'options': {
    'source': EuropeWeather.objects.all(),
    'categories': ['country', 'state']},
  'terms': {
    'avg_temp': Avg('temperature')}},
```

```
 {'options': {
     'source': AsiaWeather.objects.all(),
     'categories': ['country', 'state']},
  'terms': {
    'avg_temp': Avg('temperature')}}]
```

The second `avg_temp` will overwrite the first one. Instead just use different names for each of the keys in all the dictionaries. Here is the **right** format.

```
[{'options': {
    'source': EuropeWeather.objects.all(),
    'categories': ['country', 'state']},
  'terms': {
    'europe_avg_temp': Avg('temperature')}},
 {'options': {
     'source': AsiaWeather.objects.all(),
     'categories': ['country', 'state']},
  'terms': {
    'asia_avg_temp': Avg('temperature')}}]
```

11.1.2 How to create the charts

Chart

`Chart.__init__`(*datasource*, *series_options*, *chart_options=None*, *x_sortf_mapf_mts=None*)

Chart accept the datasource and some options to create the chart and creates it.

Arguments:

- **datasource** (**required**) - a `DataPool` object that holds the terms and other information to plot the chart from.
- **series_options** (**required**) - specifies the options to plot the terms on the chart. It is of the form

```
[{'options': {
    #any items from HighChart series. For ex.,
    'type': 'column'
   },
   'terms': {
     'x_name': ['y_name',
                {'other_y_name': {
                   #overriding options}},
                ...],
     ...
     },
   },
  ... #repeat dicts with 'options' & 'terms'
  ]
```

Where -

  - **options** (**required**) - a `dict`. Any of the parameters from the Highcharts options object - series array are valid as entries in the `options` dict except `data` (because data array is generated from your datasource by chartit). For example, `type`, `xAxis`, etc. are all valid entries here.

Note: The items supplied in the options dict are not validated to make sure that Highcharts actually supports them. Any invalid options are just passed to Highcharts JS which silently ignores them.

– **terms** (**required**) - a `dict`. keys are the x-axis terms and the values are lists of y-axis terms for that particular x-axis term. Both x-axis and y-axis terms must be present in the corresponding datasource, otherwise an APIInputError is raised.

The entries in the y-axis terms list must either be a `str` or a `dict`. If entries are dicts, the keys need to be valid y-term names and the values need to be any options to override the default options. For example,

```
[{'options': {
    'type': 'column',
    'yAxis': 0},
  'terms': {
    'city': [
      'temperature',
     {'rainfall': {
        'type': 'line',
        'yAxis': 1}}]}}]
```

plots a column chart of city vs. temperature as a line chart on yAxis: 0 and city vs. rainfall as a line chart on yAxis: 1. This can alternatively be expressed as two separate entries:

```
[{'options': {
    'type': 'column',
    'yAxis': 0},
  'terms': {
    'city': [
      'temperature']}},
 {'options': {
    'type': 'line',
    'yAxis': 1},
  'terms': {
    'city': [
      'rainfall']}}]
```

- **chart_options** (*optional*) - a `dict`. Any of the options from the Highcharts options object are valid (except the options in the `series` array which are passed in the `series_options` argument. The following `chart_options` for example, set the chart title and the axes titles.

```
{'chart': {
   'title': {
     'text': 'Weather Chart'}},
 'xAxis': {
   'title': 'month'},
 'yAxis': {
   'title': 'temperature'}}
```

Note: The items supplied in the `chart_options` dict are not validated to make sure that Highcharts actually supports them. Any invalid options are just passed to Highcharts JS which silently ignores them.

Raises:

- `APIInputError` if any of the terms are not present in the corresponding datasource or if the `series_options` cannot be parsed.

PivotChart

`PivotChart.__init__` (*datasource*, *series_options*, *chart_options=None*)

Creates the PivotChart object.

Arguments:

- **datasource** (**required**) - a `PivotDataPool` object that holds the terms and other information to plot the chart from.
- **series_options** (**required**) - specifies the options to plot the terms on the chart. It is of the form

```
[{'options': {
    #any items from HighChart series. For ex.
    'type': 'column'
    },
  'terms': [
    'a_valid_term',
    'other_valid_term': {
      #any options to override. For ex.
     'type': 'area',
      ...
      },
    ...
    ]
  },
  ... #repeat dicts with 'options' & 'terms'
  ]
```

Where -

– **options** (**required**) - a `dict`. Any of the parameters from the Highcharts options object - series array are valid as entries in the `options` dict except `data` (because data array is generated from your datasource by chartit). For example, `type`, `xAxis`, etc. are all valid entries here.

Note: The items supplied in the options dict are not validated to make sure that Highcharts actually supports them. Any invalid options are just passed to Highcharts JS which silently ignores them.

– **terms** (**required**) - a `list`. Only terms that are present in the corresponding datasource are valid.

Note: All the `terms` are plotted on the `y-axis`. The **categories of the datasource are plotted on the x-axis. There is no option to override this.**

Each of the `terms` must either be a `str` or a `dict`. If entries are dicts, the keys need to be valid terms and the values need to be any options to override the default options. For example,

```
[{'options': {
    'type': 'column',
    'yAxis': 0},
  'terms': [
    'temperature',
    {'rainfall': {
```

```
        'type': 'line',
        'yAxis': 1}}]}]
```

plots a pivot column chart of temperature on yAxis: 0 and a line pivot chart of rainfall on yAxis: 1. This can alternatively be expressed as two separate entries:

```
[{'options': {
    'type': 'column',
    'yAxis': 0},
  'terms': [
      'temperature']},
 {'options': {
    'type': 'line',
    'yAxis': 1},
  'terms': [
      'rainfall']}]
```

- **chart_options** (*optional*) - a `dict`. Any of the options from the Highcharts options object are valid (except the options in the `series` array which are passed in the `series_options` argument. The following `chart_options` for example, set the chart title and the axes titles.

```
{'chart': {
   'title': {
     'text': 'Weather Chart'}},
 'xAxis': {
   'title': 'month'},
 'yAxis': {
   'title': 'temperature'}}
```

Note: The items supplied in the `chart_options` dict are not validated to make sure that Highcharts actually supports them. Any invalid options are just passed to Highcharts JS which silently ignores them.

Raises:

- `APIInputError` if any of the terms are not present in the corresponding datasource or if the `series_options` cannot be parsed.

11.1.3 How to use chartit django template filters

`chartit.templatetags.chartit.`**`load_charts`**(*chart_list=None*, *render_to=''*)

Loads the `Chart`/`PivotChart` objects in the `chart_list` to the HTML elements with id's specified in `render_to`.

Arguments

- **chart_list** - a list of Chart/PivotChart objects. If there is just a single element, the Chart/PivotChart object can be passed directly instead of a list with a single element.
- **render_to** - a comma separated string of HTML element id's where the charts needs to be rendered to. If the element id of a specific chart is already defined during the chart creation, the `render_to` for that specific chart can be an empty string or a space.

  For example, `render_to = 'container1, , container3'` renders three charts to three locations in the HTML page. The first one will be rendered in the HTML element with id `container1`,

the second one to it's default location that was specified in `chart_options` when the Chart/PivotChart object was created, and the third one in the element with id `container3`.

Returns

- a JSON array of the HighCharts Chart options. Also returns a link to the `chartloader.js` javascript file to be embedded in the webpage. The `chartloader.js` has a jQuery script that renders a HighChart for each of the options in the JSON array

11.1.4 Quick Reference for `series` and `series_options`

PivotDataPool series	PivotChart series_options

`[{'options': {`
`  'source': SomeModel.objects.all(),`
`  'top_n_per_cat': 10, ...`
`  }`
`  'terms': {`
`  'any_name_here': Sum('a_valid_field`
`↪'),`
`  'some_other_name':{`
`  'func': Avg('a_valid_field),`
`  #any options to override`
`  ...`
`  },`
`  ...`
`  }`
`},`
`... #repeat dicts with 'options' &`
`↪'terms'`
`]`
	`[{'options': {`
	`  #any items from HighChart series.␣`
	`↪For ex.`
	`  'type': 'column'`
	`  },`
	`  'terms': [`
	`  'a_valid_term',`
	`  'other_valid_term': {`
	`  #any options to override. For ex.`
	`  'type': 'area',`
	`  ...`
	`  },`
	`  ...`
	`  ]`
	`},`
	`... #repeat dicts with 'options' &`
	`↪'terms'`
	`]`

DataPool series	Chart series_options

`[{'options': {`
`  'source': SomeModel.objects.all(),`
 	
`  }`
`  'terms': [`
`  'a_valid_field_name',`
`  ..., # more valid field names`
`  {'any_name': 'a_valid_field_name',`
`  ... # more name:field_name pairs`
`  },`
`  ]`
`},`
`... #repeat dicts with 'options' &`
`↪'terms'`
`]`
	`[{'options': {`
	`  #any items from HighChart series.␣`
	`↪For ex.`
	`  'type': 'column'`
	`  },`
	`  'terms': {`
	`  'x_name': ['y_name', 'y_name', ...],`
	`  # only corresponding keys from␣`
	`↪DataPool`
	`  # terms are valid names.`
	`  ...`
	
	`  }`
	`},`
	`... #repeat dicts with 'options' &`
	`↪'terms'`
	`]`

CHAPTER 12

Full module documentation

12.1 chartit

12.1.1 chartit package

Subpackages

chartit.highcharts package

Submodules

chartit.highcharts.hcoptions module

Module contents

chartit.templatetags package

Submodules

chartit.templatetags.chartit module

Implements the {% load_charts %} template tag!

`chartit.templatetags.chartit.`**`json_serializer`**(*obj*)
Return JSON representation of some special data types.

`chartit.templatetags.chartit.`**`load_charts`**(*chart_list=None*, *render_to=''*)
Loads the `Chart`/`PivotChart` objects in the `chart_list` to the HTML elements with id's specified in `render_to`.

Arguments

- **chart_list** - a list of Chart/PivotChart objects. If there is just a single element, the Chart/PivotChart object can be passed directly instead of a list with a single element.
- **render_to** - a comma separated string of HTML element id's where the charts needs to be rendered to. If the element id of a specific chart is already defined during the chart creation, the `render_to` for that specific chart can be an empty string or a space.

  For example, `render_to = 'container1, , container3'` renders three charts to three locations in the HTML page. The first one will be rendered in the HTML element with id `container1`, the second one to it's default location that was specified in `chart_options` when the Chart/PivotChart object was created, and the third one in the element with id `container3`.

  Returns

- a JSON array of the HighCharts Chart options. Also returns a link to the `chartloader.js` javascript file to be embedded in the webpage. The `chartloader.js` has a jQuery script that renders a HighChart for each of the options in the JSON array

Module contents

Submodules

chartit.chartdata module

class chartit.chartdata.**DataPool**(*series*)

Bases: `object`

DataPool holds the data retrieved from various models (tables).

__dict__ = mappingproxy({'_generate_vqs': <function DataPool._generate_vqs>, '__modul

__init__(*series*)

Create a DataPool object as specified by the `series`.

Arguments

- **series** *(list of dict)* - specifies the what data to retrieve and where to retrieve it from. It is of the form

```
[{'options': {
   'source': a django model, Manager or QuerySet,
   },
 'terms': [
   'a_valid_field_name', ... ,
   {'any_name': 'a_valid_field_name', ... },
   ]
},
...
]
```

  Where

  - **options** (**required**) - a `dict`. Any of the series options for the Highcharts `options` object are valid.
  - **terms** - is a list. Each element in `terms` is either
    1. a `str` - needs to be a valid model field for the corresponding `source`, or
    2. a `dict` - need to be of the form `{'any_name': 'a_valid_field_name', ...}`.

To retrieve data from multiple models or QuerySets, just add more dictionaries with the corresponding `options` and terms.

Raises

- **APIInputError** - sif the `series` argument has any invalid parameters.

> **Warning:** All elements in `terms` **must be unique** across all the dictionaries in the `series` list. If there are two terms with same `name`, the latter one is going to overwrite the one before it.

For example, the following is **wrong**:

```
[{'options': {
    'source': SomeModel},
  'terms':[
    'foo',
    'bar']},
 {'options': {
    'source': OtherModel},
  'terms':[
    'foo']}]
```

In this case, the term `foo` from `OtherModel` is going to **overwrite** `foo` from `SomeModel`.

Here is the **right** way of retrieving data from two different models both of which have the same field name.

```
[{'options': {
   'source': SomeModel},
  'terms':[
    'foo',
    'bar']},
 {'options': {
   'source': OtherModel},
  'terms':[
    {'foo_2': 'foo'}]}]
```

__module__ = 'chartit.chartdata'

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

_generate_vqs()

_get_data()

_group_terms_by_query(*sort_by_term=None*, **addl_grp_terms*)
Groups all the terms that can be extracted in a single query. This reduces the number of database calls.

Returns

- a list of sub-lists where each sub-list has items that can all be retrieved with the same query (i.e. terms from the same source and any additional criteria as specified in addl_grp_terms).

class chartit.chartdata.**PivotDataPool**(*series*, *top_n_term=None*, *top_n=None*, *pareto_term=None*, *sortf_mapf_mts=None*)

Bases: *chartit.chartdata.DataPool*

PivotDataPool holds the data retrieved from various tables (models) and then *pivoted* against the category fields.

__init__(*series*, *top_n_term=None*, *top_n=None*, *pareto_term=None*, *sortf_mapf_mts=None*)
Creates a PivotDataPool object.

Arguments

- **series** (**required**) - a list of dicts that specifies the what data to retrieve, where to retrieve it from and how to pivot the data. It is of the form

```
[{'options': {
    'source': django Model, Manager or QuerySet ,
    'categories': ['a_valid_field', ...],
    'legend_by': ['a_valid_field', ...] (optional),
    'top_n_per_cat': a number (optional),
  },
  'terms': {
    'any_name_here': django Aggregate,
    'some_other_name':{
      'func': django Aggregate,
      #any options to override
      ...
    },
  ...
  }
 },
 ... #repeat dicts with 'options' & 'terms'
]
```

Where

  - **options** - is a dict that specifies the common options for all the terms.
    - **source** (**required**) - is either a `Model`, `Manager` or a `QuerySet`.
    - **categories** (**required**) - is a list of model fields by which the data needs to be pivoted by. If there is only a single item, `categories` can just be a string instead of a list with single element.

      For example if you have a model with `country`, `state`, `county`, `city`, `date`, `rainfall`, `temperature` and you want to pivot the data by `country` and `state`, then `categories = ['country', 'state']`.

      Note: Order of elements in the `categories` list matters!

      `categories = ['country', 'state']` groups your data first by `country` and then by `state` when running the SQL query. This obviously is not the same as grouping by `state` first and then by `country`.
    - **legend_by** (*optional*) - is a list of model fields by which the data needs to be legended by. For example, in the above case, if you want to legend by `county` and `city`, then `legend_by = ['county', 'city']`

      Note: Order of elements in the `legend_by` list matters!

      See the note in `categories` above.

    * **top_n_per_cat** (*optional*) - The number of top items that the legended entries need to be limited to in each category. For example, in the above case, if you wanted only the top 3 `county/cities` with highest rainfall for each of the `country/state`, then `top_n_per_cat = 3`.

  – **terms** - is a `dict`. The keys can be any strings (but helps if they are meaningful aliases for the field). The values can either be

    * a django `Aggregate` : of a valid field in corresponding model. For example, `Avg('temperature')`, `Sum('price')`, etc. or

    * a `dict`: In this case the `func` must specify relevant django aggregate to retrieve. For example `'func': Avg('price')`. The dict can also have any additional entries from the options dict. Any entries here will override the entries in the `options` dict.

- **top_n_term** (*optional*) - a string. Must be one of the keys in the corresponding `terms` in the `series` argument.

- **top_n** (*optional*) - an integer. The number of items for the corresponding `top_n_term` that need to be retained.

  If `top_n_term` and `top_n` are present, only the `top_n` numberof items are going to displayed in the pivot chart. For example, if you want to plot only the top 5 states with highest average rainfall, you can do something like this.

```
PivotDataPool(
  series = [
     {'options': {
        'source': RainfallData.objects.all(),
        'categories': 'state'},
      'terms': {
        'avg_rain': Avg('rainfall')}}],
  top_n_term = 'avg_rain',
  top_n = 5)
```

  Note that the `top_n_term` is `'avg_rain'` and **not** `state`; because we want to limit by the average rainfall.

- **pareto_term** (*optional*) - the term with respect to which the pivot chart needs to be paretoed by.

  For example, if you want to plot the average rainfall on the y-axis w.r.t the state on the x-axis and want to pareto by the average rainfall, you can do something like this.

```
PivotDataPool(
  series = [
     {'options': {
        'source': RainfallData.objects.all(),
        'categories': 'state'},
      'terms': {
        'avg_rain': Avg('rainfall')}}],
  pareto_term = 'avg_rain')
```

- **sortf_mapf_mts** (*optional*) - a `tuple` with three elements of the form `(sortf, mapf, mts)` where

  – **sortf** - is a function (or a callable) that is used as a *key* when sorting the category values.

    For example, if `categories = 'month_num'` and if the months need to be sorted in reverse order, then `sortf` can be

```
sortf = lambda *x: (-1*x[0],)
```

Note: `sortf` is passed the category values as tuples and must return tuples!

If `categories` is `['city', 'state']` and if the category values returned need to be sorted with state first and then city, then `sortf` can be

```
sortf = lambda *x: (x[1], x[0])
```

The above `sortf` is passed tuples like `('San Francisco', 'CA')`, `('New York', 'NY')`, `...` and it returns tuples like `('CA', 'San Francisco')`, `('NY', 'New York')`, `...` which when used as keys to sort the category values will obviously first sort by state and then by city.

– **mapf** - is a function (or a callable) that defines how the category values need to be mapped.

For example, let's say `categories` is `'month_num'` and that the category values that are retrieved from your database are `1`, `2`, `3`, etc. If you want month *names* as the category values instead of month numbers, you can define a `mapf` to transform the month numbers to month names like so

```
def month_name(*t):
    names ={1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr',
            5: 'May', 6: 'Jun', 7: 'Jul', 8: 'Aug',
            9: 'Sep', 10: 'Oct', 11: 'Nov', 12: 'Dec'}
    month_num = t[0]
    return (names[month_num], )

mapf = month_name
```

Note: `mapf` like `sortf` is passed the category values as tuples and must return tuples.

– **mts** - *map then sort* ; a `bool`. If `True`, the category values are mapped first and then sorted, and if `False` category values are sorted first and then mapped.

In the above example of month names, we `mts` must be `False` because the months must first be sorted based on their number and then mapped to their names. If `mts` is `True`, the month numbers would be transformed to the month names, and then sorted, which would yield an order like `Apr`, `Aug`, `Dec`, etc. (not what we want).

Raises

- **APIInputError** - if the `series` argument has any invalid parameters.

Here is a full example of a `series` term that retrieves the average temperature of the top 3 cities in each country/state and the average rainfall of the top 2 cities in each country/state.

```
[{'options': {
    'source': Weather.objects.all(),
    'categories': ['country', 'state'],
    'legend_by': 'city',
    'top_n_per_cat': 3},
  'terms': {
    'avg_temp': Avg('temperature'),
```

```
    'avg_rain': {
      'func': Avg('rainfall'),
      'top_n_per_cat': 2}}}]
```

The `'top_n_per_cat': 2` term in `avg_rain` dict overrides `'top_n_per_cat': 5` from the comon options dict. Effectively, the above `series` retrieves the *top 2* `cities` with highest `avg_rain` in each `country/state` and *top 3* `cities` with highest `avg_temp` in each `country/state`.

A single `PivotDataPool` can hold data from multiple Models. If there are more models or QuerySets to retrieve the data from, just add more dicts to the series list with different `source` values.

Warning: The `keys` for the `terms` must be **unique across all the dictionaries** in the `series` list! If there are multiple terms with same `key`, the latter ones will just overwrite the previous ones.

For instance, the following example is **wrong**.

```
[{'options': {
    'source': EuropeWeather.objects.all(),
    'categories': ['country', 'state']},
  'terms': {
    'avg_temp': Avg('temperature')}},
 {'options': {
     'source': AsiaWeather.objects.all(),
     'categories': ['country', 'state']},
  'terms': {
    'avg_temp': Avg('temperature')}}]
```

The second `avg_temp` will overwrite the first one. Instead just use different names for each of the keys in all the dictionaries. Here is the **right** format.

```
[{'options': {
    'source': EuropeWeather.objects.all(),
    'categories': ['country', 'state']},
  'terms': {
    'europe_avg_temp': Avg('temperature')}},
 {'options': {
     'source': AsiaWeather.objects.all(),
     'categories': ['country', 'state']},
  'terms': {
    'asia_avg_temp': Avg('temperature')}}]
```

__module__ = 'chartit.chartdata'

_generate_vqs()
Generates and yields the value query set for each query in the query group.

_get_data()

chartit.charts module

class chartit.charts.**BaseChart**
Bases: `object`

Common ancestor class for all charts to avoid code duplication.

__dict__ = mappingproxy({'__weakref__': <attribute '__weakref__' of 'BaseChart' objec

__init__()

__module__ = 'chartit.charts'

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

to_json()
Load Chart's data as JSON Useful in Ajax requests. Example:

Return JSON from this method and response to client:

```
return JsonResponse(cht.to_json(), safe=False)
```

Then use jQuery load data and create Highchart:

```
$(function(){
$.getJSON("/data",function(data){
    $('#container').highcharts(JSON.parse(data));
    });
});
```

class chartit.charts.**Chart**(*datasource*, *series_options*, *chart_options=None*, *x_sortf_mapf_mts=None*)
Bases: *chartit.charts.BaseChart*

__init__(*datasource*, *series_options*, *chart_options=None*, *x_sortf_mapf_mts=None*)
Chart accept the datasource and some options to create the chart and creates it.

Arguments:

- **datasource** (**required**) - a `DataPool` object that holds the terms and other information to plot the chart from.
- **series_options** (**required**) - specifies the options to plot the terms on the chart. It is of the form

```
[{'options': {
    #any items from HighChart series. For ex.,
    'type': 'column'
   },
   'terms': {
     'x_name': ['y_name',
                {'other_y_name': {
                   #overriding options}},
                ...],
     ...
     },
   },
  ... #repeat dicts with 'options' & 'terms'
  ]
```

Where -

– **options** (**required**) - a `dict`. Any of the parameters from the Highcharts options object - series array are valid as entries in the `options` dict except `data` (because data array is generated from your datasource by chartit). For example, `type`, `xAxis`, etc. are all valid entries here.

Note: The items supplied in the options dict are not validated to make sure that Highcharts actually supports them. Any invalid options are just passed to Highcharts JS which silently ignores them.

– **terms** (**required**) - a `dict`. keys are the x-axis terms and the values are lists of y-axis terms for that particular x-axis term. Both x-axis and y-axis terms must be present in the corresponding datasource, otherwise an APIInputError is raised.

The entries in the y-axis terms list must either be a `str` or a `dict`. If entries are dicts, the keys need to be valid y-term names and the values need to be any options to override the default options. For example,

```
[{'options': {
    'type': 'column',
    'yAxis': 0},
  'terms': {
    'city': [
      'temperature',
     {'rainfall': {
        'type': 'line',
        'yAxis': 1}}]}}]
```

plots a column chart of city vs. temperature as a line chart on yAxis: 0 and city vs. rainfall as a line chart on yAxis: 1. This can alternatively be expressed as two separate entries:

```
[{'options': {
    'type': 'column',
    'yAxis': 0},
  'terms': {
    'city': [
      'temperature']}},
 {'options': {
    'type': 'line',
    'yAxis': 1},
  'terms': {
    'city': [
      'rainfall']}}]
```

- **chart_options** (*optional*) - a `dict`. Any of the options from the Highcharts options object are valid (except the options in the `series` array which are passed in the `series_options` argument. The following `chart_options` for example, set the chart title and the axes titles.

```
{'chart': {
   'title': {
     'text': 'Weather Chart'}},
 'xAxis': {
   'title': 'month'},
 'yAxis': {
   'title': 'temperature'}}
```

Note: The items supplied in the `chart_options` dict are not validated to make sure that Highcharts actually supports them. Any invalid options are just passed to Highcharts JS which silently ignores them.

Raises:

- `APIInputError` if any of the terms are not present in the corresponding datasource or if the `series_options` cannot be parsed.

`__module__ = 'chartit.charts'`

_groupby_x_axis_and_vqs()
Here is an example of what this function would return

```
{
    0: {
        0: {'month_seattle': ['seattle_temp']},
        1: {'month': ['houston_temp', 'boston_temp']}
    }
}
```

In the above example,

- the inner most dict keys ('month' and 'month_seattle') are on the same xAxis (xAxis 0), just groupped in 2 groups (0 and 1)
- the inner most list values are from same ValueQuerySet (table)

If you decide to display multiple chart types with multiple axes then the return value will look like this

```
{
    0: {
        0: {'month': ['boston_temp']}
    },
    1: {
        0: {'month': ['houston_temp']}
    }
}
```

- the outer most 0 and 1 are the numbers of the x axes
- the inner most 0 shows that each axis has 1 data group

_set_default_hcoptions(*chart_options*)
Set some default options, like xAxis title, yAxis title, chart title, etc.

generate_plot()

class chartit.charts.**PivotChart**(*datasource*, *series_options*, *chart_options=None*)
Bases: *chartit.charts.BaseChart*

__init__(*datasource*, *series_options*, *chart_options=None*)
Creates the PivotChart object.

Arguments:

- **datasource** (**required**) - a `PivotDataPool` object that holds the terms and other information to plot the chart from.
- **series_options** (**required**) - specifies the options to plot the terms on the chart. It is of the form

```
[{'options': {
    #any items from HighChart series. For ex.
    'type': 'column'
    },
  'terms': [
    'a_valid_term',
    'other_valid_term': {
      #any options to override. For ex.
     'type': 'area',
      ...
      },
```

```
    ...
    ]
  },
  ... #repeat dicts with 'options' & 'terms'
  ]
```

Where -

– **options** (**required**) - a `dict`. Any of the parameters from the Highcharts options object - series array are valid as entries in the `options` dict except `data` (because data array is generated from your datasource by chartit). For example, `type`, `xAxis`, etc. are all valid entries here.

Note: The items supplied in the options dict are not validated to make sure that Highcharts actually supports them. Any invalid options are just passed to Highcharts JS which silently ignores them.

– **terms** (**required**) - a `list`. Only terms that are present in the corresponding datasource are valid.

Note: All the `terms` are plotted on the `y-axis`. The **categories of the datasource are plotted on the x-axis. There is no option to override this.**

Each of the `terms` must either be a `str` or a `dict`. If entries are dicts, the keys need to be valid terms and the values need to be any options to override the default options. For example,

```
[{'options': {
    'type': 'column',
    'yAxis': 0},
  'terms': [
    'temperature',
    {'rainfall': {
        'type': 'line',
        'yAxis': 1}}]}]
```

plots a pivot column chart of temperature on yAxis: 0 and a line pivot chart of rainfall on yAxis: 1. This can alternatively be expressed as two separate entries:

```
[{'options': {
    'type': 'column',
    'yAxis': 0},
  'terms': [
     'temperature']},
 {'options': {
    'type': 'line',
    'yAxis': 1},
  'terms': [
     'rainfall']}]
```

- **chart_options** (*optional*) - a `dict`. Any of the options from the Highcharts options object are valid (except the options in the `series` array which are passed in the `series_options` argument. The following `chart_options` for example, set the chart title and the axes titles.

```
{'chart': {
   'title': {
     'text': 'Weather Chart'}},
```

```
'xAxis': {
  'title': 'month'},
'yAxis': {
  'title': 'temperature'}}
```

Note: The items supplied in the `chart_options` dict are not validated to make sure that Highcharts actually supports them. Any invalid options are just passed to Highcharts JS which silently ignores them.

Raises:

- `APIInputError` if any of the terms are not present in the corresponding datasource or if the `series_options` cannot be parsed.

__module__ = 'chartit.charts'

generate_plot()

set_default_hcoptions()

chartit.exceptions module

Global ChartIt exception and warning classes.

exception chartit.exceptions.**APIInputError**
Bases: Exception

Some kind of problem when validating the user input.

__module__ = 'chartit.exceptions'

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

chartit.utils module

utility and helper functions.

class chartit.utils.**RecursiveDefaultDict**(*data=None*)
Bases: dict

Behaves exactly the same as a collections.defaultdict but works with pickle.loads. Fixes #10.

__dict__ = mappingproxy({'__weakref__': <attribute '__weakref__' of 'RecursiveDefault

__getitem__(*key*)

__init__(*data=None*)

__module__ = 'chartit.utils'

__setitem__(*key*, *item*)

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

update(*element*)

chartit.utils.**_convert_to_rdd**(*obj*)
Accepts a dict or a list of dicts and converts it to a RecursiveDefaultDict.

chartit.utils.**_getattr**(*obj*, *attr*)
Recurses through an attribute chain to get the ultimate value.

chartit.validation module

Validates input parameters.

chartit.validation.**_clean_categories**(*categories*, *source*)

chartit.validation.**_clean_field_aliases**(*fa_actual*, *fa_cat*, *fa_lgby*)

chartit.validation.**_clean_legend_by**(*legend_by*, *source*)

chartit.validation.**_clean_source**(*source*)

chartit.validation.**_convert_cso_to_dict**(*series_options*)

chartit.validation.**_convert_dps_to_dict**(*series_list*)

chartit.validation.**_convert_pcso_to_dict**(*series_options*)

chartit.validation.**_convert_pdps_to_dict**(*series_list*)

chartit.validation.**_validate_field_lookup_term**(*model*, *term*, *query*)
Checks whether the term is a valid field_lookup for the model.

Args:

- **model** (**required**) - a django model for which to check whether the term is a valid field_lookup.
- **term** (**required**) - the term to check whether it is a valid field lookup for the model supplied.
- **query** - the source query so we can check for aggregate or extra fields.

Returns:

- The verbose name of the field if the supplied term is a valid field.

Raises:

- APIInputError: If the term supplied is not a valid field lookup parameter for the model.

chartit.validation.**_validate_func**(*func*)

chartit.validation.**_validate_top_n_per_cat**(*top_n_per_cat*)

chartit.validation.**clean_cso**(*series_options*, *ds*)
Clean the Chart series_options input from the user.

chartit.validation.**clean_dps**(*series*)
Clean the DataPool series input from the user.

chartit.validation.**clean_pcso**(*series_options*, *ds*)
Clean the PivotChart series_options input from the user.

chartit.validation.**clean_pdps**(*series*)
Clean the PivotDataPool series input from the user.

chartit.validation.**clean_sortf_mapf_mts**(*sortf_mapf_mts*)

chartit.validation.**clean_x_sortf_mapf_mts**(*x_sortf_mapf_mts*)

chartit.validation.**get_all_field_names**(*meta*)
Taken from Django 1.9.8 b/c this is unofficial API which has been deprecated in 1.10.

Module contents

This Django application can be used to create charts and pivot charts directly from models.

CHAPTER 13

License

- Copyright (c) 2011, Praveen Gollakota.
- Copyright (c) 2016, Grant McConnaughey.
- Copyright (c) 2016, Alexander Todorov.
- and other contributors.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

Python Module Index

C

Index

Symbols

A

B

C

D

G

J

L

P

R

S

T

U