django-report-tools Documentation

Release 0.2.1

Evan Brumley

Jul 20, 2017

Contents

Report tools aims to take the pain out of putting charts, graphs and tables into your Django projects. It lets you do the following:

- Define your reports using the same syntax as Django forms and models
- Use built-in 'renderers' to avoid the hassle of dealing with various charting technologies (currently only the Google Visualization Toolkit is supported)
- Enter chart data in a standardised format
- Build a simple API, allowing for the creation of chart exports or a 'save to dashboard' feature.

An example report:

```python
class MyReport(Report):
    renderer = GoogleChartsRenderer

    pie_chart = charts.PieChart(
        title="A nice, simple pie chart",
        width=400,
        height=300
    )

    def get_data_for_pie_chart(self):
        data = ChartData()

        data.add_column("Pony Type")
        data.add_column("Population")

        data.add_row(["Blue", 20])
        data.add_row(["Pink", 20])
        data.add_row(["Magical", 1])

        return data
```

For an expanation of this code, read on to the *getting started* section.

CHAPTER 1

Contents

Getting Started

Overview

This library deals with the following concepts:

Report A class that maintains a collection of charts

Chart A class that, given a set of options and a set of data, generates HTML to display that data in a nice format. Charts can include graphs, tables, and other arbitrary visualization types.

ChartData A class that allows the collection of data in a structured format, which can then be passed into a Chart.

ChartRenderer A class that implements the rendering of Chart objects. Changing the ChartRenderer class for a Report or Chart allows for easy changing of rendering technologies - i.e. google charts, FusionCharts, HighCharts etc.

ReportView A class-based view to assist with the creation of a report API. More information in the *API documentation*.

Report objects

Consider the following, very simple report - this would usually be located in a `reports.py` file in your app directory:

```
# myapp/reports.py

from report_tools.reports import Report
from report_tools.chart_data import ChartData
from report_tools.renderers.googlecharts import GoogleChartsRenderer
from report_tools import charts


class MyReport(Report):
    renderer = GoogleChartsRenderer

    pie_chart = charts.PieChart(
        title="A nice, simple pie chart",
        width=400,
        height=300
    )

    def get_data_for_pie_chart(self):
        data = ChartData()

        data.add_column("Pony Type")
        data.add_column("Population")

        data.add_row(["Blue", 20])
        data.add_row(["Pink", 20])
        data.add_row(["Magical", 1])

        return data
```

A `Report` is composed of `Chart` objects. In this case, the report has a single chart `pie_chart`. This chart has been given a title, width and height, but other than that it will use the default rendering options.

For information on the available `Chart` classes, see the *Chart documentation*.

Rendering for this report will be performed by the `GoogleChartsRenderer` class, which uses Google's visualization framework. For more information on the available rendering classes, see the *renderer documentation*.

All charts on a report require a `get_data_for_xxx` method, where `xxx` is the attribute name given to the chart. In this case, you can see the `get_data_for_pie_chart` field has been created. Pie charts require a `ChartData` object as input, so the `get_data_for_pie_chart` method creates one, fills it with data and returns it. For detailed information on how to enter data into a `ChartData` object or other data storage formats, see the *ChartData documentation*.

Using a report in your views

Using a report in your view is simple:

```
# myapp/views.py

from django.shortcuts import render
from myapp.reports import MyReport


def my_report(request):
    # Initialise the report
    template = "myapp/my_report.html"
    report = MyReport()
    context = {'report': report}

    return render(request, template, context)
```

You can also use the included class based view, which will help if you wish to use API features down the track:

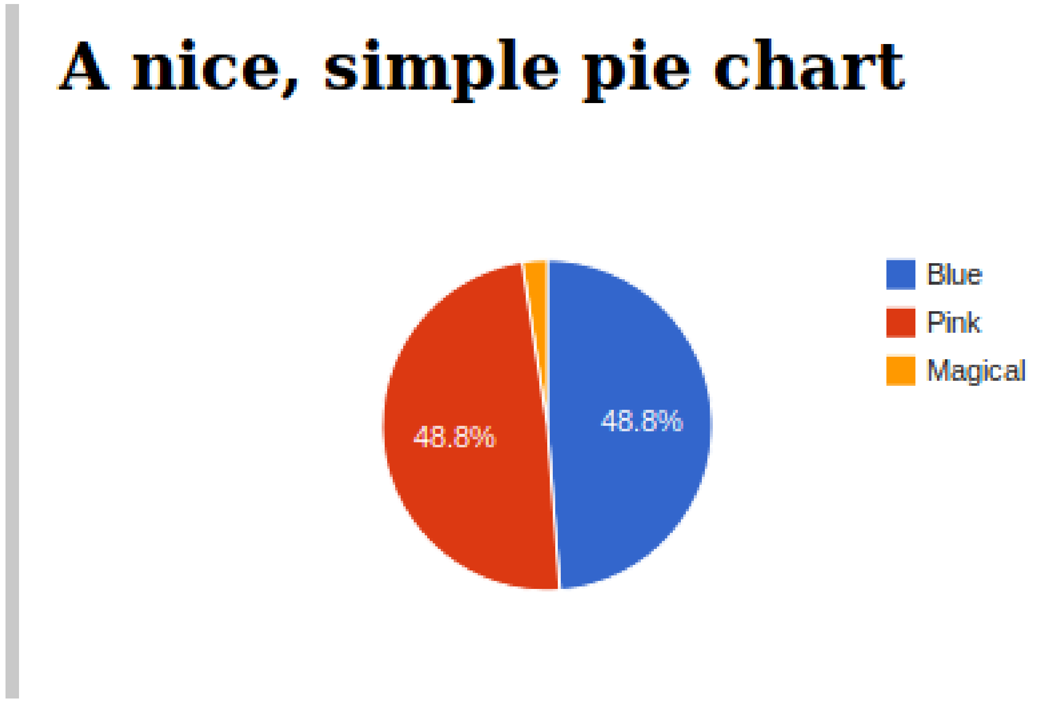
```
# myapp/views.py

from django.shortcuts import render
from myapp.reports import MyReport
from report_tools.views import ReportView


class MyReportView(ReportView):
    def get_report(self, request):
        return MyReport()

    def get(self, request):
        template = 'myapp/my_report.html'
        context = {
            'report': self.get_report(request),
        }

        return render(request, template, context)
```

Note the use of the `get_report` method. This provides the hook required for the API to grab the report. For more information, check the *API documentation*.

Using a report in your templates

Using a report in your template is also straightforward

```
<!-- templates/myapp/my_report.html -->

<h1 class="chart-header">{{ report.pie_chart.title }}</h1>

<div class="chart-container">
        {{ report.pie_chart }}
</div>
```



Charts

class `report_tools.charts.`**`Chart`**`(`***kwargs*`)`

Charts represent a view of your data. Charts are usually a graph of some sort, but may also contain arbitrary HTML (see the TemplateChart class)

Optional Chart Arguments

title

`Chart.`**`title`**

A "human-friendly" title for your chart. Note that this field will typically not be rendered in the chart itself - it is provided for use in the surrounding HTML. For example:

```
<h2 class="chart-title">{{ my_report.chart.title }}</h2>
<div class="chart-container">{{ my_report.chart }}</div>
```

If you want to embed a title in the chart itself, you should refer to the *renderer documentation* for your chosen rendering engine.

renderer

`Chart.`**`renderer`**

If you want the chart to use a different renderer to the one specified on the report, you can use this to pass in the appropriate renderer class.

renderer_options

`Chart.`**`renderer_options`**

Renderers will typically have a lot of specific customization options. This argument accepts a dictionary, which will be passed through to the renderer. For available options, check out the *renderer documentation*.

attrs

`Chart.`**`attrs`**

If you want to store some extra information with the chart (i.e. what sort of container div should it use?), you can pass in a dictionary of attributes with this argument.

Built-in Chart Classes

PieChart

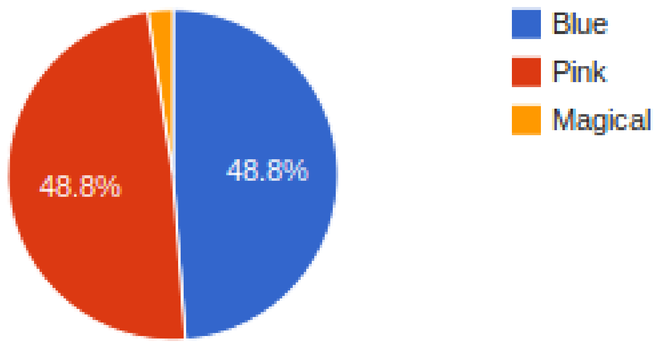
class `report_tools.charts.`**`PieChart`**`(`*width=None*, *height=None*, ***kwargs*`)`

A standard pie chart. The corresponding `get_data_for_xxx` method should provide a `ChartData` object with two columns. Column one should contain the data point labels, and column 2 should contain numerical values.

Accepts two extra keyword arguments, *width* and *height*. These can be integers, floats or strings, depending on what your chosen rendering engine supports.

Example:

```
class MyReport(Report):
    renderer = GoogleChartsRenderer

    pie_chart = charts.PieChart(width=400, height=300)

    def get_data_for_pie_chart(self):
        data = ChartData()

        data.add_column("Pony Type")
        data.add_column("Population")

        data.add_row(["Blue", 20])
        data.add_row(["Pink", 20])
        data.add_row(["Magical", 1])

        return data
```



ColumnChart

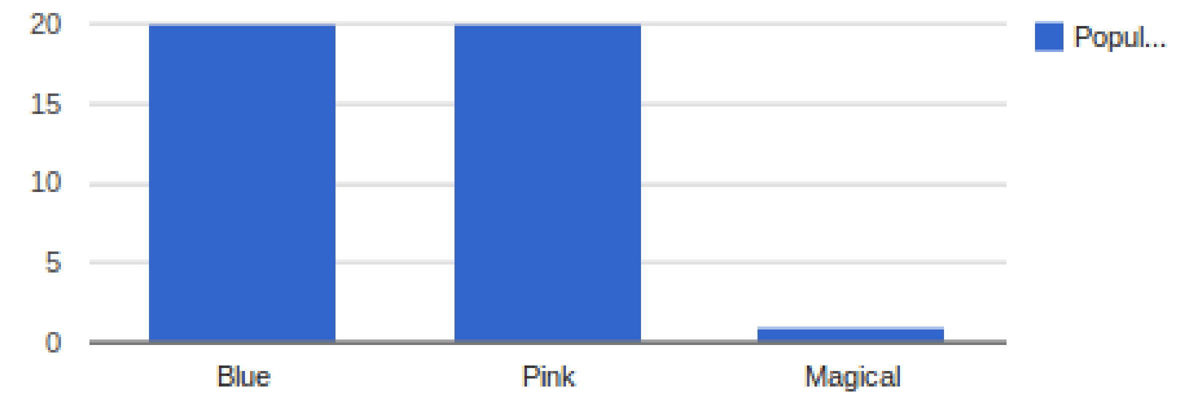
class `report_tools.charts.`**`ColumnChart`**(*width=None*, *height=None*, ***kwargs*)

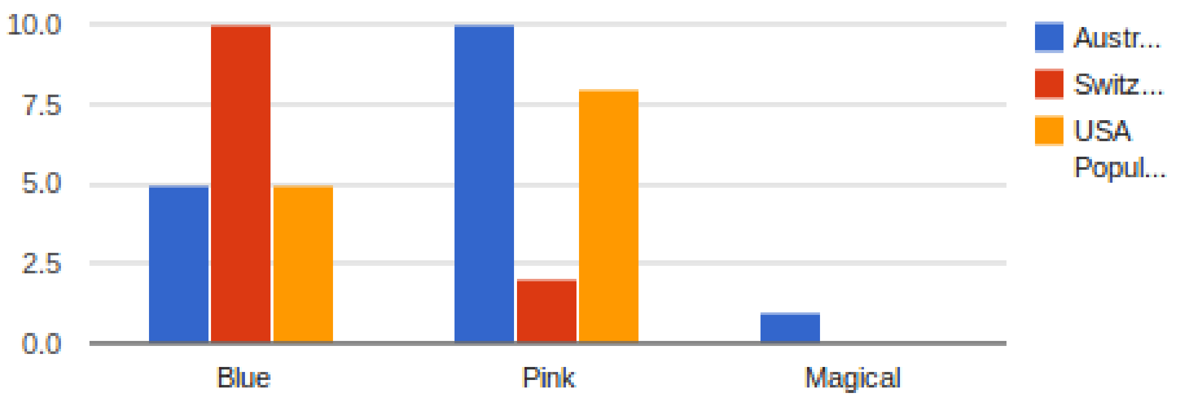
A standard vertical column chart. The corresponding `get_data_for_xxx` method should provide a ChartData object with 1+n columns, where n is the number of data series to be displayed. Column one should contain the data point labels, and subsequent columns should contain numerical values.

Accepts two extra keyword arguments, *width* and *height*. These can be integers, floats or strings, depending on what your chosen rendering engine supports.

Example:

```
class MyReport(Report):
    renderer = GoogleChartsRenderer

    column_chart = charts.ColumnChart(title="Pony Populations", width="500")
    multiseries_column_chart = charts.ColumnChart(title="Pony Populations by Country",
↪ width="500")

    def get_data_for_column_chart(self):
        data = ChartData()

        data.add_column("Pony Type")
        data.add_column("Population")

        data.add_row(["Blue", 20])
        data.add_row(["Pink", 20])
        data.add_row(["Magical", 1])

        return data

    def get_data_for_multiseries_column_chart(self):
        data = ChartData()

        data.add_column("Pony Type")
        data.add_column("Australian Population")
        data.add_column("Switzerland Population")
        data.add_column("USA Population")

        data.add_row(["Blue", 5, 10, 5])
        data.add_row(["Pink", 10, 2, 8])
        data.add_row(["Magical", 1, 0, 0])

        return data
```





BarChart

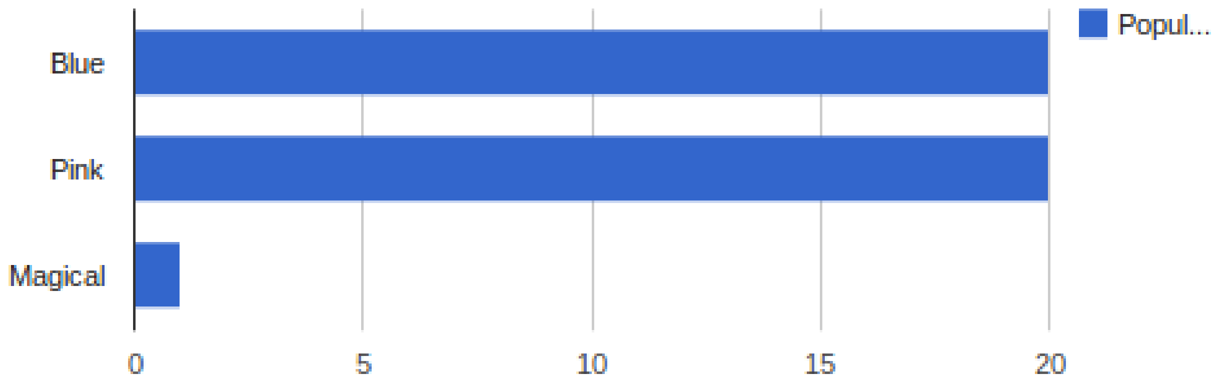
class `report_tools.charts.`**`BarChart`**(*width=None*, *height=None*, ***kwargs*)

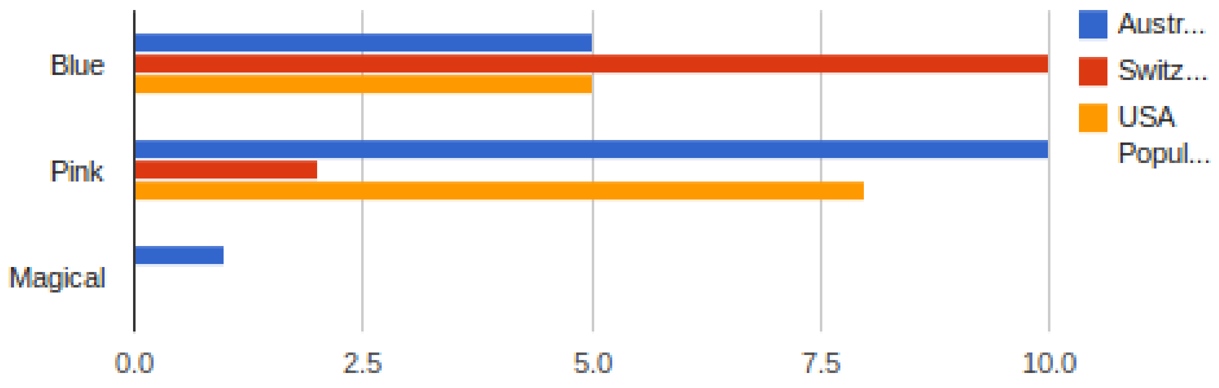
A standard horizontal bar chart. The corresponding `get_data_for_xxx` method should provide a ChartData object with 1+n columns, where n is the number of data series to be displayed. Column one should contain the data point labels, and subsequent columns should contain numerical values.

Accepts two extra keyword arguments, *width* and *height*. These can be integers, floats or strings, depending on what your chosen rendering engine supports.

Example:

```
class MyReport(Report):
    renderer = GoogleChartsRenderer

    bar_chart = charts.BarChart(title="Pony Populations", width="500")
    multiseries_bar_chart = charts.BarChart(title="Pony Populations by Country",␣
↪width="500")

    def get_data_for_bar_chart(self):
        data = ChartData()

        data.add_column("Pony Type")
        data.add_column("Population")

        data.add_row(["Blue", 20])
        data.add_row(["Pink", 20])
        data.add_row(["Magical", 1])

        return data

    def get_data_for_multiseries_bar_chart(self):
        data = ChartData()

        data.add_column("Pony Type")
        data.add_column("Australian Population")
        data.add_column("Switzerland Population")
        data.add_column("USA Population")

        data.add_row(["Blue", 5, 10, 5])
        data.add_row(["Pink", 10, 2, 8])
        data.add_row(["Magical", 1, 0, 0])

        return data
```





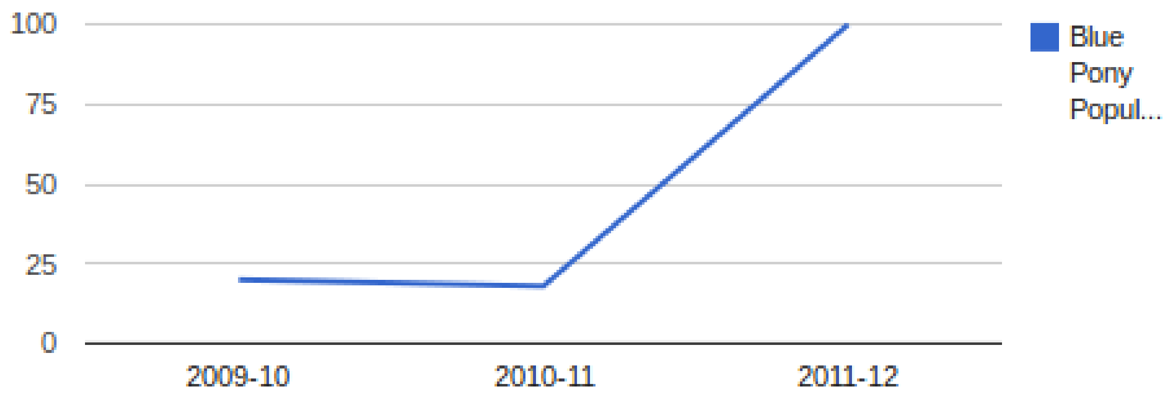
LineChart

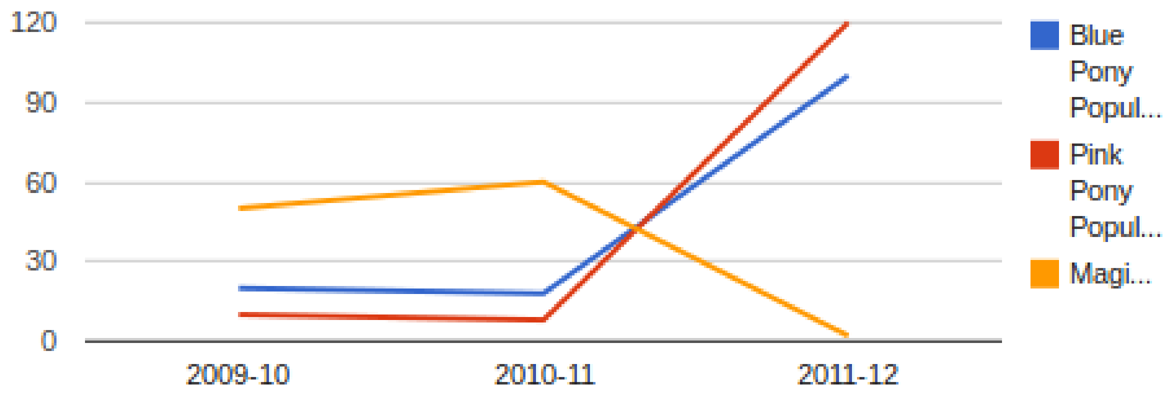
class `report_tools.charts.`**`LineChart`**(*width=None*, *height=None*, ***kwargs*)

A standard line chart. The corresponding `get_data_for_xxx` method should provide a ChartData object with 1+n columns, where n is the number of data series to be displayed. Column one should contain the data point labels, and subsequent columns should contain numerical values.

Example:

```
class MyReport(Report):
    renderer = GoogleChartsRenderer

    line_chart = charts.LineChart(title="Blue Pony Population - 2009-2012", width="500
↪")
    multiseries_line_chart = charts.LineChart(title="Pony Populations - 2009-2012",␣
↪width="500")

    def get_data_for_line_chart(self):
        data = ChartData()

        data.add_column("Test Period")
        data.add_column("Blue Pony Population")

        data.add_row(["2009-10", 20])
        data.add_row(["2010-11", 18])
        data.add_row(["2011-12", 100])

        return data

    def get_data_for_multiseries_line_chart(self):
        data = ChartData()

        data.add_column("Test Period")
        data.add_column("Blue Pony Population")
        data.add_column("Pink Pony Population")
        data.add_column("Magical Pony Population")

        data.add_row(["2009-10", 20, 10, 50])
        data.add_row(["2010-11", 18, 8, 60])
        data.add_row(["2011-12", 100, 120, 2])

        return data
```





TemplateChart

class `report_tools.charts.`**`TemplateChart`**(*template*, ***kwargs*)

This chart simply renders a given template. The `get_data_for_xxx` method should return a dictionary context. An extra context variable 'chart_id' will be provided, which should be used if a unique identifier is required in the template. Note that the template chart does not require a renderer.

Accepts one required argument, *template*.

Example:

```
class MyReport(Report):
    template_chart = charts.TemplateChart(template="myapp/template_chart.html")

    def get_data_for_template_chart(self):
        pony_types = [
            ('Blue', 'Equus Caeruleus'),
            ('Pink', 'Equus Roseus'),
            ('Magical', 'Equus Magica')
        ]

        template_context = {
            'pony_types': pony_types
        }

        return template_context
```

```
<!-- myapp/template_chart.html -->

<table id="{{ chart_id }}" border="1">
    <thead>
        <th>Pony Type</th>
        <th>Latin Name</th>
    </thead>
    <tbody>
        {% for pony_type, latin_name in pony_types %}
            <tr>
                <td>{{ pony_type }}</td>
                <td>{{ latin_name }}</td>
            </tr>
        {% endfor %}
    </tbody>
</table>
```

Chart Renderers

Chart renderers control the way reports display on your site. Currently, the only included renderer is for Google Charts, but more are on the way, and it's easy to write one for your own favourite charting package.

Included Renderers

Google Charts

class `report_tools.renderers.googlecharts.`**`GoogleChartsRenderer`**

The google charts renderer uses Google Chart Tools to render the built-in chart types.

Chart Support

The google chart renderer supports all the built-in chart types described in the *chart documentation*. This includes:

- Pie Charts
- Column Charts
- Multi-series Column Charts
- Bar Charts
- Multi-series Bar Charts
- Line Charts
- Multi-series Line Charts

Extra Charts

There are currently no additional chart types included with the google charts renderer, although support for table charts and geo charts is planned.

Prerequisites

To use the google charts renderer, you must import the google javascript api by including the following html in your page:

```
<script type="text/javascript" src="https://www.google.com/jsapi"></script>
```

Renderer Options

The `renderer_options` dictionary for charts using the google charts renderer is simply JSON encoded and passed directly into the chart initialization javascript. You therefore have full control over any parameter defined in the Google Chart Tools documentation:

- Google Chart Tools Pie Chart Documentation
- Google Chart Tools Column Chart Documentation
- Google Chart Tools Bar Chart Documentation
- Google Chart Tools Line Chart Documentation

For example, if you want to create a stacked column chart with no legend, a light grey background and red and blue columns, your chart definition might look something like the following:

```
class MyReport(Report):
    renderer = GoogleChartsRenderer

    stacked_column_chart = charts.ColumnChart(
        title="Pony Populations",
        width="500",
        renderer_options={
            'isStacked': True,
            'legend': {
                'position': 'none',
            },
            'backgroundColor': '#f5f5f5',
            'series': [
                {'color': '#ff0000'},
                {'color': '#0000ff'},
            ],
        }

    )

    def get_data_for_stacked_column_chart(self):
        ...
```

Tips and Tricks

If you need to override the default html/javascript that the google renderer creates, you can override the default templates at:

- `report_tools/renderers/googlecharts/barchart.html`
- `report_tools/renderers/googlecharts/columnchart.html`
- `report_tools/renderers/googlecharts/linechart.html`
- `report_tools/renderers/googlecharts/piechart.html`

Basic Usage

Renderers are typically defined on your report objects. For example:

```python
class MyReport(Report):
    renderer = GoogleChartsRenderer

    column_chart = charts.ColumnChart(title="Pony Populations", width="500")

    def get_data_for_column_chart(self):
        ...
```

You can also select renderers on a chart-by-chart basis. For example:

```python
class MyReport(Report):
    renderer = GoogleChartsRenderer

    column_chart = charts.ColumnChart(title="Pony Populations", width="500")

    column_chart_other_renderer = charts.ColumnChart(
        title="Pony Populations",
        width="500",
        renderer=SomeOtherRenderer
    )

    def get_data_for_column_chart(self):
        ...

    def get_data_for_column_chart_other_renderer(self):
        ...
```

Talking to Your Renderer

Above and beyond the basic options described in the *chart documentation*, individual renderers usually provide a lot of unique customization options. You can set these by passing in a `renderer_options` dictionary to the chart. For example, for a red background using the Google Charts renderer:

```python
class MyReport(Report):
    renderer = GoogleChartsRenderer

    column_chart = charts.ColumnChart(
        title="Pony Populations",
        width="500",
        renderer_options={
            'backgroundColor': "#ff0000"
        }
    )

    def get_data_for_column_chart(self):
        ...
```

For information on the the various options available, refer to the documentation for your chosen renderer above.

Writing Your Own

A very simple stub of a chart renderer looks something like the following:

```
from report_tools.renderers import ChartRenderer


class MyChartRenderer(ChartRenderer):
    @classmethod
    def render_piechart(cls, chart_id, options, data, renderer_options):
        return "<div id='%s' class='placeholder'>Pie Chart</div>" % chart_id

    @classmethod
    def render_columnchart(cls, chart_id, options, data, renderer_options):
        return "<div id='%s' class='placeholder'>Column Chart</div>" % chart_id

    @classmethod
    def render_barchart(cls, chart_id, options, data, renderer_options):
        return "<div id='%s' class='placeholder'>Bar Chart</div>" % chart_id

    @classmethod
    def render_linechart(cls, chart_id, options, data, renderer_options):
        return "<div id='%s' class='placeholder'>Line Chart</div>" % chart_id
```

When a chart is rendered, it goes to the selected chart renderer class and tries to call an appropriate class method. This method will typically be named `render_xxx` where `xxx` is a lower case representation of the chart's class name. All rendering methods take the same parameters:

chart_id A unique identifier for the chart. Safe for use as an html element id.

options If a chart accepts additional parameters, such as width, height or template, they will be loaded into this dictionary.

data The data returned by the chart's `get_data_for_xxx` method. This typically comes in as a ChartData object, so you'll need to wrangle it into something your charting package can read.

renderer_options The renderer options specified when the chart was defined on the report.

Using ChartData to fill your charts

class `report_tools.chart_data.`**`ChartData`**

The ChartData class provides a consistent way to get data into charts. Each ChartData object represents a 2 dimensional array of cells, which can be annotated on a column, row or cell level.

`add_column`(*self*, *name*, *metadata=None*)

Adds a new column to the data table.

Parameters

- **`name`** – The name of the column
- **`metadata`** – A dictionary of metadata describing the column

`add_columns` (*self*, *columns*)
Adds multiple columns to the data table

Parameters **`columns`** – A list of column names. If you need to enter metadata with the columns, you can also pass in a list of name-metadata tuples.

`report_tools.chart_data.`**`add_row`** (*self*, *data*, *metadata=None*)
Adds a new row to the datatable

Parameters

- **`data`** – A list of data points that will form the row. The length of the list should match the number of columns added.
- **`metadata`** – A dictionary of metadata describing the row

`report_tools.chart_data.`**`add_rows`** (*self*, *rows*)
Adds multiple rows to the data table

Parameters **`rows`** – A list of rows. If you need to enter metadata with the rows, you can also pass in a list of row-metadata tuples.

`get_columns` (*self*)
Returns a list of columns added so far

..method:: get_rows(self)

Returns a list of rows added so far

Creating an API

Sometimes it’s not enough just to have your charts accessible within the context of a larger report. Sometimes you need to pull them out, pass them around and so on. The following steps should provide a good way to start off a more complex data reporting system.

Step 1 - Use the class-based view

The API relies on structured, class-based views to provide the hooks necessary to generate the reports and charts. The following is an example:

```python
# myapp/views.py

from django.shortcuts import render
from myapp.reports import MyReport
from report_tools.views import ReportView


class MyReportView(ReportView):
    def get_report(self, request):
        return MyReport()

    def get(self, request):
        template = 'myapp/my_report.html'
        context = {
            'report': self.get_report(request),
        }

        return render(request, template, context)
```

This is a really simple class-based view. It behaves in the same way as Django's base *View* class, with the addition of a *get_report* method. This method provides the necessary hook for the API to extract the report without touching your other view code.

Step 2 - Register the class-based view

To register the view with the api, simply pass it into the `register` function in the `api` module, along with the key you wish to use to access the report later:

```
from report_tools.views import ReportView
from report_tools.api import register


class MyReportView(ReportView):
    ...

register(MyReportView, 'myreportview_api_key')
```

Step 3 (optional) - Add the API endpoints to your urls.py

If you plan to make your chart HTML available externally, you can let the API handle your URLS for you by adding the following line to your *urls.py*.

```
url(r'^api/', include('report_tools.urls'))
```

Access a chart internally

To access a chart from a registered report, simply use the `report_tools.api.get_chart` function.

`report_tools.api.`**`get_chart`**(*request*, *api_key*, *chart_name*, *parameters=None*, *prefix=None*)

Parameters

- **`request`** – The current request
- **`api_key`** – The API key used to register the report view
- **`chart_name`** – The attribute name given to the required chart
- **`parameters`** – If provided, this dictionary will override the GET parameters in the provided request.
- **`prefix`** – If provided, this string will be prepended to the chart's id. Useful if you're displaying the same chart from the same report with different parameters.

Returns The requested chart object

Access a chart externally

If you've added the API to your urls.py (step 3), you should be able to access a simple JSON endpoint at `api/report_api_key/chart_name/`. The endpoint will provide the chart HTML along with a dictionary of anything supplied in the chart's *attrs* parameter.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index