svg.charts

Release 4.1.dev0+g0d33396.d20170409

Apr 09, 2017

Contents

CHAPTER 1

charts Package

svg Package

svg is a namespace package, meant to be shared with other libraries implementing svg functionality.

svg.charts Package

svg.charts is the package containing the various modules of this library.

bar Module

class svg.charts.bar.**Bar**(*fields*, **args*, ***kargs*)

Bases: *svg.charts.graph.Graph*

Create presentation quality SVG bar graphs easily.

Synopsis:

```
from svg.charts import bar


fields = 'Jan Feb Mar'.split()
data_sales_02 = [12, 45, 21]

bc = bar.VerticalBar(fields, {'width': 300, 'height': 500})

bc.add_data({'data': data_sales_02, 'title': 'Sales 2002'})

print("Content-type: image/svg+xml\r\n\r\n")
print(bc.burn())
```

Description

This object aims to allow you to easily create high quality SVG bar graphs. You can either use the default style sheet or supply your own. Either way there are many options which can be configured to give you control over how the graph is generated - with or without a key, data elements at each point, title, subtitle etc.

Notes

The default stylesheet handles upto 12 data sets, if you use more you must create your own stylesheet and add the additional settings for the extra data sets. You will know if you go over 12 data sets as they will have no style and be in black.

Examples

See the example usage in tests/samples.py

See also

- svg.charts.graph
- svg.charts.line
- svg.charts.pie
- svg.charts.plot
- svg.charts.time_series

bar_gap = True

data_max()

data_min()

data_range()

get_bar_gap(*field_size*)

get_data_labels()

get_data_values()

get_field_labels()

scale_divisions = None

stack = 'overlap'

stylesheet_names = ['graph.css', 'bar.css']

class svg.charts.bar.**VerticalBar**(*fields*, **args*, ***kargs*)
Bases: *svg.charts.bar.Bar*

draw_data()

get_x_labels()

get_y_labels()

top_align = 1

top_font = 1

x_label_offset(*width*)

class svg.charts.bar.**HorizontalBar**(*fields*, **args*, ***kargs*)
Bases: *svg.charts.bar.Bar*

draw_data()

get_x_labels()

get_y_labels()

right_align = True

right_font = True

rotate_y_labels = True

show_x_guidelines = True

show_y_guidelines = False

y_label_offset(*height*)

css Module

graph Module

svg.charts.graph

The base module for *svg.charts* classes.

class svg.charts.graph.**DrawHooks**

Bases: object

Mix-in for Graph subclasses providing hooks at various points in the rendering of a Graph.

Use with any Graph subclass like so:

```
class MyVerticalBar(DrawHooks, VerticalBar):
        def after_draw_data(self):
                self.root.append(...)
```

after_draw_data()

before_draw_data()

draw_data()

class svg.charts.graph.**Graph**(*config={}*)

Bases: object

Base object for generating SVG Graphs

Synopsis

> This class is only used as a superclass of specialized charts. Do not attempt to use this class directly, unless creating a new chart type.
>
> For examples of how to subclass this class, see the existing specific subclasses, such as svn.charts.Pie.

- svg.charts.bar
- svg.charts.line
- svg.charts.pie
- svg.charts.plot
- svg.charts.time_series

KEY_BOX_SIZE = 12

add_data(*conf*)
Add data to the graph object. May be called several times to add additional data sets.

conf should be a dictionary including 'data' and 'title' keys

add_defs(*defs*)
Override and place code to add defs here. TODO: what are defs?

add_popup(*x*, *y*, *label*)
Add pop-up information to a point on the graph.

burn()
Process the template with the data and config which has been set and return the resulting SVG.

Raises ValueError when no data set has been added to the graph object.

calculate_bottom_margin()
Calculate the margin in pixels below the plot area, setting border_bottom.

calculate_graph_dimensions()

calculate_left_margin()
Calculates the margin to the left of the plot area, setting border_left.

calculate_offsets_bottom()

calculate_right_margin()
Calculate the margin in pixels to the right of the plot area, setting border_right.

calculate_top_margin()
Calculate the margin in pixels above the plot area, setting border_top.

clear_data()
This method removes all data from the object to create a new graph with the same config options.

css_inline = False

draw_graph()
The central logic for drawing the graph.

Sets self.graph (the 'g' element in the SVG root)

draw_graph_subtitle()

draw_graph_title()

draw_legend()

draw_titles()
Draws the graph title and subtitle

draw_x_guidelines(*label_height*, *count*)
Draw the X-axis guidelines

draw_x_label(*label*)

draw_x_labels()
Draw the X axis labels

draw_x_title()

draw_y_guidelines(*label_height*, *count*)
Draw the Y-axis guidelines

draw_y_label(*label*)

draw_y_labels()
Draw the Y axis labels

draw_y_title()

field_height()

field_width()

font_size = 12

get_field_height()

get_field_width()

get_stylesheet()

get_stylesheet_resources()
Get the stylesheets for this instance

get_y_offset()

graph_subtitle = 'Graph Subtitle'

graph_title = 'Graph Title'

height = 300

key = True

key_font_size = 10

key_position = 'right'

keys()

load_config(*config*)

static load_resource_stylesheet(*name*, *subs={}*)

make_datapoint_text(*x*, *y*, *value*, *style=None*)
Add text for a datapoint

max_y_label_width_px()
Calculate the width of the widest Y label. This will be the character height if the Y labels are rotated.

min_scale_value = None

parse_css()
Take a .css file (classes only please) and parse it into a dictionary of class/style pairs.

process_data(*data*)

static render(*tree*)

render_inline_styles()
Hard-code the styles into the SVG XML if style sheets are not used.

right_align = 0

right_font = 0

rotate_x_labels = False

rotate_y_labels = False

scale_integers = False

show_data_values = True

show_graph_subtitle = False

show_graph_title = False

show_x_guidelines = False

show_x_labels = True

show_x_title = False

show_y_guidelines = True

show_y_labels = True

show_y_title = False

stagger_x_labels = False

stagger_y_labels = False

start_svg()
Base SVG Document Creation

step_include_first_x_label = True

step_include_first_y_label = True

step_x_labels = 1

step_y_labels = 1

stylesheet_names = ['graph.css']

subtitle_font_size = 14

title_font_size = 16

top_align = 0

top_font = 0

validate_data(*conf*)

width = 500

x_label_font_size = 12

x_label_offset(*width*)
Return an offset for drawing the x label. Currently returns 0.

x_title = 'X Field names'

x_title_font_size = 14

y_label_font_size = 12

y_label_offset(*height*)
Return an offset for drawing the y label. Currently returns 0.

y_offset

y_title = 'Y Scale'

y_title_font_size = 14

y_title_text_direction = 'bt'

class svg.charts.graph.**class_dict**(*obj*)
Bases: object

Emulates a dictionary, but retrieves class attributes

keys()

line Module

class svg.charts.line.**Line**(*config={}*)
Bases: *svg.charts.graph.Graph*

Line Graph

area_fill = False

calc_coords(*field*, *value*, *width=None*, *height=None*)

calculate_left_margin()

draw_data()

get_cumulative_data()
Get the data as it will be charted. The first set will be the actual first data set. The second will be the sum of the first and the second, etc.

get_x_labels()

get_y_label_values()

get_y_labels()

max_value()

min_value()

right_align = True

right_font = True

scale_divisions = None

show_data_points = True

show_data_values = True

stacked = False

stylesheet_names = ['graph.css', 'plot.css']

top_align = True

top_font = True

pie Module

class svg.charts.pie.**Pie**(*config={}*)
Bases: *svg.charts.graph.Graph*

A presentation-quality SVG pie graph

Synopsis:

```
from svg.charts.pie import Pie
fields = ['Jan', 'Feb', 'Mar']

data_sales_02 = [12, 45, 21]

graph = Pie(dict(
        height = 500,
        width = 300,
        fields = fields))
graph.add_data({'data': data_sales_02, 'title': 'Sales 2002'})
print "Content-type" image/svg+xml
```

‘

print graph.burn()

Description

This object aims to allow you to easily create high quality SVG pie graphs. You can either use the default style sheet or supply your own. Either way there are many options which can be configured to give you control over how the graph is generated - with or without a key, display percent on pie chart, title, subtitle etc.

add_data(*data_descriptor*)

Add a data set to the graph

```
>>> graph.add_data({data:[1,2,3,4]})
```

Note that a ‘title’ key is ignored.

Multiple calls to add_data will sum the elements, and the pie will display the aggregated data. e.g.

```
>>> graph.add_data({data:[1,2,3,4]})
>>> graph.add_data({data:[2,3,5,7]})
```

is the same as:

```
>>> graph.add_data({data:[3,5,8,11]})
```

If data is added of with differing lengths, the corresponding values will be assumed to be zero.

```
>>> graph.add_data({data:[1,2,3,4]})
>>> graph.add_data({data:[5,7]})
```

is the same as:

```
>>> graph.add_data({data:[5,7]})
>>> graph.add_data({data:[1,2,3,4]})
```

and

```
>>> graph.add_data({data:[6,9,3,4]})
```

add_defs(*defs*)

Add svg definitions

datapoint_font_size = 12

draw_data()

draw_graph()
Here we don't need the graph (consider refactoring)

expand_gap = 10

expand_greatest = False

expanded = False

get_x_labels()
Okay. I'll refactor after this

get_y_labels()
Definitely consider refactoring

keys()

round(*val*, *to*)

shadow_offset = 10

show_actual_values = False

show_data_labels = False

show_key_actual_values = True

show_key_data_labels = True

show_key_percent = False

show_percent = True

show_shadow = True

show_x_labels = False

show_y_labels = False

stylesheet_names = ['graph.css', 'pie.css']

svg.charts.pie.**robust_add**(*a*, *b*)
Add numbers a and b, treating None as 0

plot Module

plot.py

class svg.charts.plot.**Plot**(*config={}*)
Bases: *svg.charts.graph.Graph*

An SVG plot of scalar data.

Synopsis:

```
from svg.charts import plot

# Data sets are x,y pairs
# Note that multiple data sets can differ in length, and that the
# data in the datasets needn't be in order; they will be ordered
# by the plot along the x-axis.
projection = [
        6, 11,  0, 5,  18, 7,  1, 11,  13, 9,  1, 2,  19, 0,  3, 13,  7, 9,
]
```

```
actual = [
        0, 18,  8, 15,  9, 4,  18, 14,  10, 2,  11, 6,  14, 12,  15, 6,
        4, 17,  2, 12,
]

p = plot.Plot(dict(
        height = 500,
        width = 300,
        key = true,
        scale_x_integers = True,
        scale_y_integerrs = True,
))

p.add_data({
        'data': projection,
        'title': 'Projected',
})

p.add_data({
        'data': actual,
        'title': 'Actual',
})

print(p.burn())
```

Description

Produces a graph of scalar data.

This object aims to allow you to easily create high quality SVG scalar plots. You can either use the default style sheet or supply your own. Either way there are many options which can be configured to give you control over how the graph is generated - with or without a key, data elements at each point, title, subtitle etc.

Examples

See the examples in tests/samples.py

Notes

The default stylesheet handles upto 10 data sets, if you use more you must create your own stylesheet and add the additional settings for the extra data sets. You will know if you go over 10 data sets as they will have no style and be in black.

Unlike the other types of charts, data sets must contain x,y pairs:

```
[1, 2]     # A data set with 1 point: (1,2)
[1,2, 5,6] # A data set with 2 points: (1,2) and (5,6)
```

add_constant_line(*value*, *label=None*, *style=None*)

area_fill = False

calculate_left_margin()

calculate_right_margin()

data_max(*axis*)

data_min(*axis*)

data_range(*axis*)

draw_data()

draw_data_points(*line*, *data_points*, *graph_points*)

draw_lines_between_points = True

field_height()

field_size(*axis*)

field_width()

format(*x*, *y*)

get_data_values(*axis*)

get_graph_points(*data_points*)

get_lpath(*points*)

get_single_axis_values(*axis*, *dataset*)
Return all the values for a single axis of the data.

get_x_labels()

get_x_values()

get_y_labels()

get_y_values()

load_transform_parameters()
Cache the parameters necessary to transform x & y coordinates

max_x_value = None

max_y_value = None

min_x_value = None

min_y_value = None

process_data(*data*)

right_align = 1

right_font = 1

scale_x_divisions
Determines the scaling for the X axis divisions.

graph.scale_x_divisions = 2

would cause the graph to attempt to generate labels stepped by 2; e.g.: 0,2,4,6,8...

scale_x_integers = False

scale_y_divisions = None

scale_y_integers = False

show_data_points = True

stacked = False

stylesheet_names = ['graph.css', 'plot.css']

top_align = 1

top_font = 1

transform_output_coordinates(*point*)

validate_data(*conf*)

validate_data_flat(*series*)

validate_data_pairs(*series*)

x_data_index = 0

x_range()

y_data_index = 1

y_range()

svg.charts.plot.**get_pairs**(*i*)

schedule Module

class svg.charts.schedule.**Schedule**(*config={}*)

Bases: *svg.charts.graph.Graph*

Represents SVG plots of scalar temporal data

Synopsis:

```
from svg.charts import schedule

# Data sets are label, start, end triples.
data1 = [
        "Housesitting", "6/17/04", "6/19/04",
        "Summer Session", "6/15/04", "8/15/04",
]

sched = schedule.Schedule(dict(
        width = 640,
        height = 480,
        graph_title = "My Schedule",
        show_graph_title = True,
        no_css = True,
        scale_x_integers = True,
        scale_y_integers = True,
        min_x_value = 0,
        min_y_value = 0,
        show_data_labels = True,
        show_x_guidelines = True,
        show_x_title = True,
        x_title = "Time",
        stagger_x_labels = True,
        stagger_y_labels = True,
        x_label_format = "%m/%d/%y",
))

sched.add_data(dict(
        data = data1,
        title = 'Data',
))

print(sched.burn())
```

Description

Produces a graph of temporal scalar data.

Examples

See tests/samples.py for an example.

Notes

The default stylesheet handles upto 10 data sets, if you use more you must create your own stylesheet and add the additional settings for the extra data sets. You will know if you go over 10 data sets as they will have no style and be in black.

Note that multiple data sets within the same chart can differ in length, and that the data in the datasets needn't be in order; they will be ordered by the plot along the X-axis.

The dates must be parseable by ParseDate, but otherwise can be any order of magnitude (seconds within the hour, or years)

add_data(*data*)

Add data to the plot:

```
# A data set with 1 point: Lunch from 12:30 to 14:00
d1 = [ "Lunch", "12:30", "14:00" ]

# A data set with 2 points: "Cats" runs from 5/11/03 to 7/15/04, and
#                           "Henry V" runs from 6/12/03 to 8/20/03
d2 = [
        "Cats",    "5/11/03", "7/15/04",
        "Henry V", "6/12/03", "8/20/03",
]

sched.add_data(dict(
        data = d1,
        title = 'Meetings',
))
sched.add_data(dict(
        data = d2,
        title = 'Plays',
))
```

Note that the data must be in time,value pairs, and that the date format may be any date that is parseable by dateutil. Also note that, in this example, we're mixing scales; the data from d1 will probably not be discernable if both data sets are plotted on the same graph, since d1 is too granular.

bar_gap = True

draw_data()

format(*x*, *y*)

get_bar_gap(*field_size*)

get_min_x_value()

get_x_labels()

get_x_values()

get_y_labels()

classmethod lookup_relativedelta_parameter(*unit_string*)

```
>>> lrp = Schedule.lookup_relativedelta_parameter
>>> lrp('Years')
'years'
>>> lrp('yr')
'years'
>>> lrp('s')
'seconds'
```

min_x_value

parse_date(*date_string*)

popup_format = '%Y-%m-%d %H:%M:%S'

process_data(*conf*)

scale_x_divisions = False

scale_x_integers = False

set_min_x_value(*value*)

stylesheet_names = ['graph.css', 'bar.css']

timescale_divisions = None

validate_data(*conf*)

x_label_format = '%Y-%m-%d %H:%M:%S'

y_label_offset(*height*)

time_series Module

class svg.charts.time_series.**Plot**(*config={}*)

Bases: *svg.charts.plot.Plot*

For creating SVG plots of scalar temporal data

Synopsis:

```
from svg.charts import time_series

# Data sets are x,y pairs
data1 = ["6/17/72", 11,  "1/11/72", 7,  "4/13/04 17:31", 11,
        "9/11/01", 9,  "9/1/85", 2,  "9/1/88", 1,  "1/15/95", 13]
data2 = ["8/1/73", 18,  "3/1/77", 15,  "10/1/98", 4,  "5/1/02", 14,
        "3/1/95", 6,  "8/1/91", 12,  "12/1/87", 6,  "5/1/84", 17,
        "10/1/80", 12]

ts = time_series.Plot(dict(
        width = 640,
        height = 480,
        graph_title = "TS Title",
        show_graph_title = True,
        no_css = True,
        key = True,
        scale_x_integers = True,
        scale_y_integers = True,
        min_x_value = 0,
```

```
        min_y_value = 0,
        show_data_labels = True,
        show_x_guidelines = True,
        show_x_title = True,
        x_title = "Time",
        show_y_title = True,
        y_title = "Ice Cream Cones",
        y_title_text_direction = 'bt',
        stagger_x_labels = True,
        x_label_format = "%m/%d/%y",
))

ts.add_data(dict(
        data = projection,
        title = 'Projected',
))

ts.add_data(dict(
        data = actual,
        title = 'Actual',
))

print(ts.burn())
```

Description

Produces a graph of temporal scalar data.

Examples

See tests/samples.py for an example.

Notes

The default stylesheet handles upto 10 data sets, if you use more you must create your own stylesheet and add the additional settings for the extra data sets. You will know if you go over 10 data sets as they will have no style and be in black.

Unlike the other types of charts, data sets must contain x,y pairs:

```
# A data set with 1 point: ("12:30", 2)
["12:30", 2]
# A data set with 2 points: ("01:00", 2) and
#                           ("14:20", 6)
["01:00", 2, "14:20", 6]
```

Note that multiple data sets within the same chart can differ in length, and that the data in the datasets needn't be in order; they will be ordered by the plot along the X-axis.

The dates must be parseable by ParseDate, but otherwise can be any order of magnitude (seconds within the hour, or years)

add_data(*data*)

Add data to the plot:

```
# A data set with 1 point: ("12:30", 2)
d1 = ["12:30", 2]

# A data set with 2 points: ("01:00", 2) and
#                           ("14:20", 6)
```

```
d2 = ["01:00", 2, "14:20", 6]

graph.add_data(
        data = d1,
        title = 'One',
)
graph.add_data(
        data = d2,
        title = 'Two',
)
```

Note that the data must be in (time, value) pairs, and the date format may be any date that is parseable by dateutil.

format (*x*, *y*)

get_min_x_value()

get_time_range (*start*, *stop*, *delta*)

get_x_labels()

get_x_timescale_division_values()

get_x_values()

min_x_value

parse_date (*date_string*)

popup_format = '%Y-%m-%d %H:%M:%S'

process_data (*data*)

set_min_x_value (*date*)

timescale_divisions = None

x_label_format = '%Y-%m-%d %H:%M:%S'

svg.charts.time_series.**fromtimestamp**()
timestamp[, tz] -> tz's local time from POSIX timestamp.

util Module

svg.charts.util.**flatten_mapping** (*mapping*)
For every key that has an __iter__ method, assign the values to a key for each.

```
>>> flatten_mapping({'ab': 3, ('c','d'): 4}) == {'ab': 3, 'c': 4, 'd': 4}
True
```

svg.charts.util.**float_range** (*start=0*, *stop=None*, *step=1*)
Much like the built-in function range, but accepts floats

```
>>> tuple(float_range(0, 9, 1.5))
(0.0, 1.5, 3.0, 4.5, 6.0, 7.5)
```

svg.charts.util.**reverse_mapping** (*mapping*)
For every key, value pair, return the mapping for the equivalent value, key pair

```
>>> reverse_mapping({'a': 'b'}) == {'b': 'a'}
True
```

CHAPTER 2

History

4.0

08 Apr 2017

- Removed support for `compress` flag on Graphs. Simply invoke `zlib.compress` on the data if needed.
- `Graph.burn` now returns the rendered SVG as a Unicode string without XML declaration and without pretty-printing. To customize the XML rendering, override or replace the `Graph.render` static method.

3.4.4

04 Jan 2017

- #17: Fix issue in subclassed Line charts.

3.4.3

23 Dec 2016

- Minor tweak to README for better rendering in PyPI.

3.4.2

23 Dec 2016

- Update docs link.

3.4.1

04 Nov 2016

- #16: Restore support for passing iterable data elements.

3.4

15 Aug 2016

Issue #14:

- `svg.charts.plot.Plot` (and hence its subclass `svg.charts.time_series.Plot`) now accept data as a sequence of pairs. Example:

```
import svg.charts.plot

g = svg.charts.plot.Plot()
g.add_data(dict(title='Example',
                data=[(1, 1), (2, 4), (3, 9)]))
```

- data point labels that are drawn when `show_data_values = True` can have their label changed from the default (which is the y-value) by giving a data item a `.text` attribute. It is convenient to used `namedtuple()` for this:

```
import svg.charts.plot
from collections import namedtuple

Datum = nametuple("Datum", 'x y text')

g = svg.charts.plot.Plot()
g.add_data(dict(title='Example',
                data=[Datum(1, 1, 'first'),
                      Datum(2, 4, 'second'),
                      Datum(3, 9, 'third')]))
```

  (in fact data items can have any extra attribute; only `.text` is used currently)

3.3.1

30 Jul 2016

- #13: Fix issues in lazy-evaluated iterators.

3.3

22 Mar 2016

- #12: The library now provides an `svg.charts.graph.DrawHooks` mix-in for customizing the draw behavior.
- Cleaned up documentation.

3.2

22 Mar 2016

- #8: Subtitle is now rendering the subtitle and not the title again.
- #9: `Bar` objects now expose a `_fill_class` method for overriding the default CSS fill class. It's now possible to create a custom subclass that generates the fill based on the field index as well. For example:

```
class VerticalBar(svg.charts.bar.VerticalBar):
    key = False

    def _fill_class(self, dataset_index, field_index):
        fill_index = 1 + dataset_index + field_index*len(self.data)
        return 'fill%s' % fill_index
```

3.1

25 Apr 2015

- Adding a couple small dependencies eliminated a lot of duplicated code in the `util` module.
- Corrected error when `stacked` was used in Line charts.

3.0

23 Apr 2015

- Dropped support for Python 2.6.
- Requires setuptools for installation.
- Filter out comments when parsing CSS.
- Corrected errors in `Graph.render_inline_styles`.

2.3

21 Jan 2014

- #4: Added hook in Graph to allow overriding of the attributes on the root SVG element. One can now override or monkeypatch `Graph._get_root_attributes` to alter the rendering of the root attributes such as width and height. For example, to omit width and height:

```
class MyPlot(plot.Plot):
    def _get_root_attributes(self):
        attrs = super(MyPlot, self)._get_root_attributes()
        del attrs['width']
        del attrs['height']
        return attrs
```

2.2.2

21 Jan 2014

- #1: Fixed javascript ID names in TimeSeries labels.

2.2.1

21 Jan 2014

- #5: Fixed references to class attributes in `graph.py`.

2.2

10 Nov 2013

- SF Issue #1: Fixed installation on Unix systems again. Author's preference for lowercase `readme.txt` was trumped by setuptools '#100 <https://bitbucket.org/pypa/setuptools/issue/100/>'_.
- Moved hosting to BitBucket.
- Established Continuous Integration Tests on Github mirror using Travis-CI.

2.1

23 Sep 2013

- Project now builds and tests pass on Python 3 without 2to3.

2.0.9

12 Oct 2011

- Corrected buggy logic in y-axis label rendering (thanks to Emmanuel Blot).
- Converted to Unix line endings.

2.0.8

31 Jul 2011

- Updated to latest cssutils with Python 3 support. Thanks Christof!
- Fixed a few remaining issues with Python 3 compatibility.

2.0.7

02 Apr 2011

- Fixed bug in rendering of Pie Chart styles.
- Improved testing framework. Now samples are at least generated as part of the test suite.
- Fixed bug in javascript when label ids had spaces. See #3139197.
- Fixed build issue where package data wasn't included due to 2to3 technique. Now using distribute technique and installation on Python 3 requires distribute.

2.0.6

27 Mar 2011

- Fixed bug where x axis labels would not be rendered properly if the largest value was the same as the largest visible x value on the chart.

2.0.5

- Altered the way CSS files are loaded, so they can be more easily customized by subclasses (and less dependent on the class names).

2.0.4

- A small attempt to improve the documentation - added links to examples that already exist.

2.0.3

- Fix IndexError in `svg.charts.plot.Plot.field_size` when there are only two values returned by float_range (in the case there are only two different 'y' values in the data) and scale_y_integers == True. Credit to Jean Schurger for the patch.
- Fixed problem in setup.py installing on Unix OS (case sensitivity of readme.txt). Credit to Luke Miller and Jean Schurger for supplying a patch for this issue.

2.0.2

- Updated cssutils dependency to 0.9.6 (currently in beta) to require the CSS profiles support.
- Completed an SVG CSS profile according to the SVG 1.1 spec.

2.0.1

- Added preliminary SVG CSS profile, suitable for stock CSS properties.

2.0

- First major divergence from the Ruby reference implementation
- Now implemented as a namespace package (svg.charts instead of svg_charts)
- Changed XML processor to lxml
- Enabled extensible css support using cssutils, greatly reducing static CSS
- Renamed modules and methods to be more consistent with PEP-8 naming convention

Upgrading from 1.x to 2.0

I suggest removing SVG 1.0 from the python installation. This involves removing the SVG directory (or svg_chart*) from site-packages.

Change import statements to import from the new namespace, so:

```
from SVG import Bar
Bar.VerticalBar(...)
```

becomes:

```
from svg.charts import bar
bar.VerticalBar(...)
```

1.2

- Bug fixes

1.1

- First public release

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

E

F

G

H

K

L

M

P

R

S

T

V

W

X

Y