FigureFirst Documentation

Release 0.1

Theodore Lindsay, Peter Weir, Floris van Breugel

Apr 15, 2022

Contents

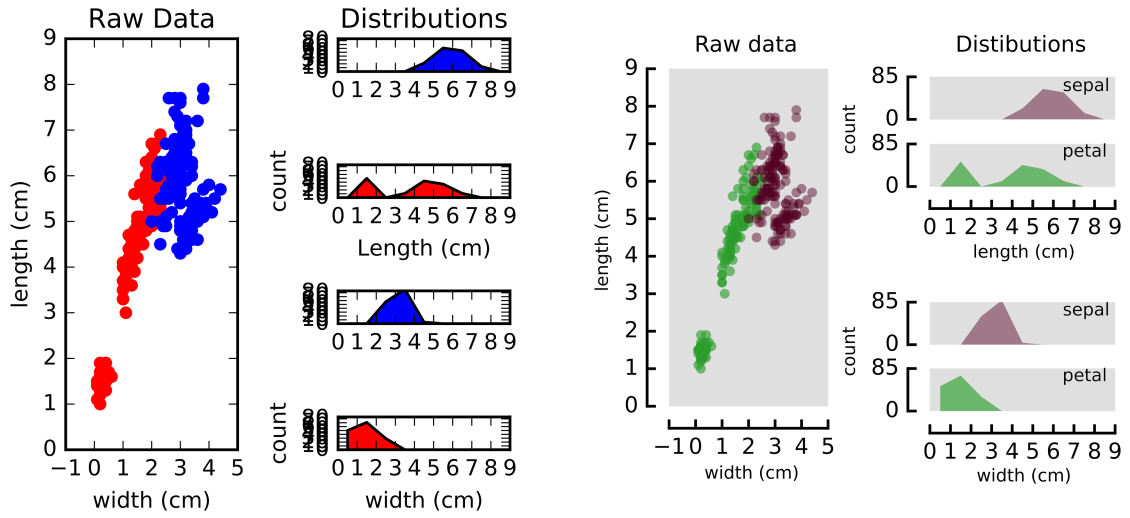
Contents:

CHAPTER 1

Introduction

1.1 What is figurefirst?

Constructing figures for publication in scientific journals requires some attention to the details of styling and layout. For example, consider the descriptive analysis of the iris dataset below. The plot on the left was constructed using the matplotlib gridspec tool, and is a perfectly acceptable representation of the data; but a more efficient use of the space can be achieved with a little manual adjustment. Note how in the version on the right, the x axis is shared across distributions, and the layout helps clarify the structure of the data. The main objective of figurefirst is to facilitate the construction of these more organic layouts with minimal work.



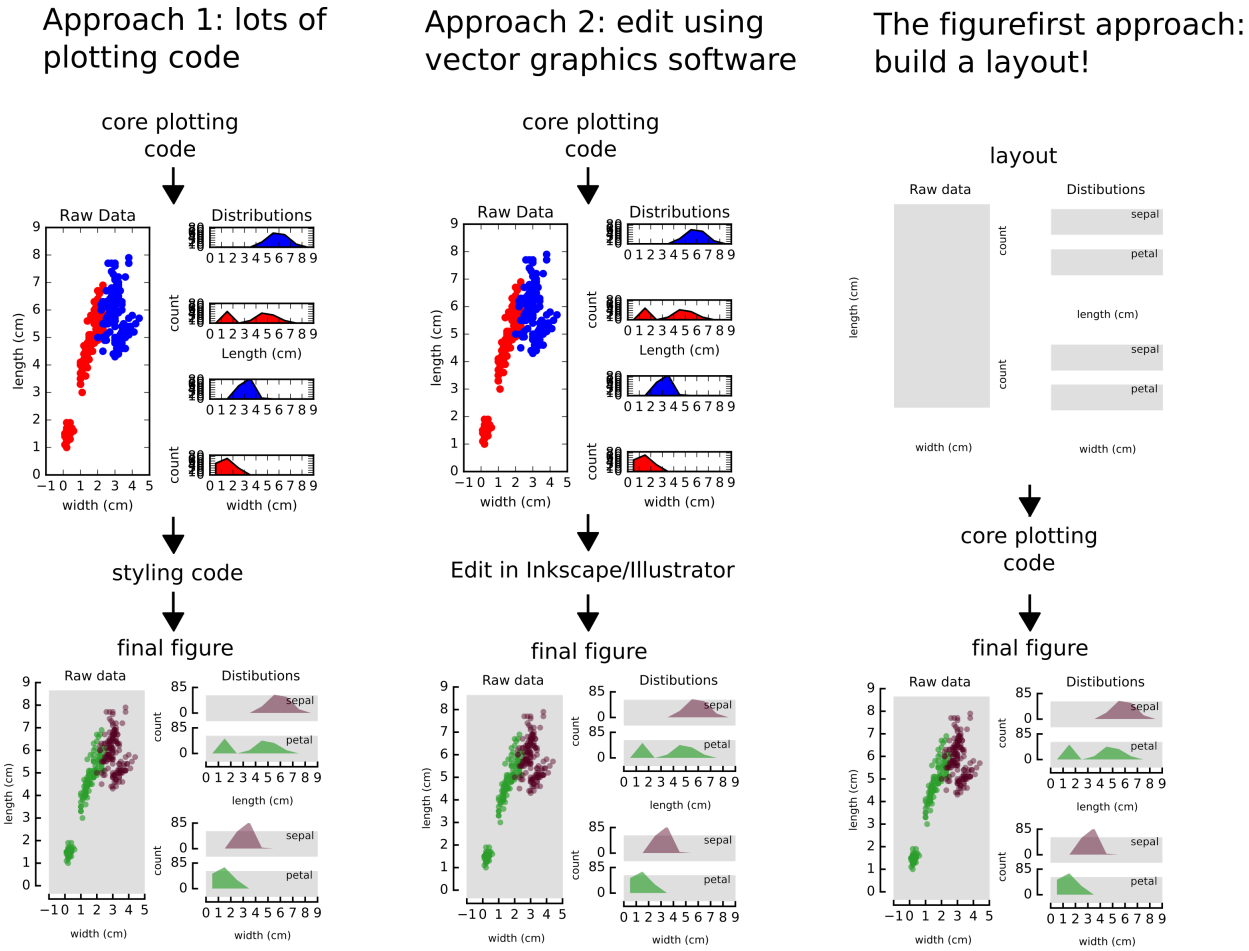
1.2 What problem does figurefirst solve?

Traditionally, there are two basic approaches for solving the flexible-layout problem.

(1) After deciding on the basic scheme, you can customize your plotting code to achieve the precise layout you want. This approach is often very time consuming because graphical decisions are difficult to evaluate using a strictly text-oriented approach. Additionally, this approach often leads to plotting functions where most of the code is devoted to styling. This makes the core plotting logic hard to parse.

(2) Construct a basic plot and then manually adjust the plots using a vector graphics tool such as Inkscape or Adobe Illustrator. This breaks down if new data need to be plotted, or the plotting function changes. For instance, after recieving reviews you may need to add a control group to your figure. You would need to re-apply all the manual adjustments in Illustator at this point.

figurefirst adds a third option: Use a layout document to specify all the stylistic decisions, and target your plotting functions to this layout as you see fit.



1.3 How does figurefirst work?

The `figurefirst` library seeks to allow effort devoted to the raw analysis and raw presentation of data to proceed *in parallel* and *independent to* the work of styling and formating the figure. The approach we take is to facilitate passing graphical information from the open-standard scalable vector graphics (svg) file format into objects consumable by

the open-source matplotlib Python plotting library. This allows authors to take advantage of sophisticated SVG editors such as Inkscape to start constructing final-form figures early in the process of manuscript preparation, while retaining the freedom to collect new data and revise analyses.

1.4 Should I use figurefirst?

The traditional work-flow may be suitable in many cases, but it is inherently a uni-directional process: an experiment is designed, data are collected and analyzed, a manuscript is written, and the analyzed data are assembled into a final figure sequence. Science rarely advances in such a linear manner: new data are collected, analyses are revised, and new experiments are conceived even well after the writing process has commenced. Some authors have proposed initiating the process of manuscript assembly at the most preliminary findings by constructing an outline and iteratively building the manuscript as experiments are performed by filling out this outline. The figurefirst philosophy is analogous, with the layout serving much like the outline, allowing the figure to be filled out as hypotheses are tested and experiments are performed.

CHAPTER 2

Installation

- To get access to the very latest features and bugfixes, clone figurefirst from github:

```
$ git clone https://github.com/FlyRanch/figurefirst.git
```

Now you can install figurefirst from the source:

```
$ python setup.py install
```

To install the Inkscape extensions you will need to copy the .inx and .py files from the `inkscape_extensions` directory to the appropriate path for inkscape extensions on your system. This is probably `~/.config/inkscape/extensions/` on Linux/OSX or `C:\Program Files\Inkscape\share\extensions` on Windows.

CHAPTER 3

Requirements

figurefirst depends on:

- Python 2.7 or Python 3.x
- numpy
- matplotlib
- dill

CHAPTER 4

Basic Usage

4.1 Overview

Creating a new figure with figurefirst generally involves four steps:

1. Design the layout file. Fundamentally this means decorating a specific subset of the objects in the svg files with xml tags that identify what objects are something figurefirst should expose to Python.
2. Convert the information in the graphical svg objects into Python objects. This is accomplished via the *FigureLayout* class.
3. Plot data, taking advantage of the objects created within the *FigureLayout* to style and organize the figure.
4. Merge newly created matplotlib figures with the original layout file and save to svg.

4.2 A multipanel figure

To get started it is worth examining how the figurefirst approach compares to generating a similar figure using matplotlib alone. Consider a five-panel figure with non-uniform axes sizes such as the example shown in the matplotlib gridspec documentation.

To generate this figure using only matplotlib and gridspec we would use the following code:

```
from matplotlib import pyplot as plt
ax1 = plt.subplot2grid((3, 3), (0, 0), colspan=3)
ax2 = plt.subplot2grid((3, 3), (1, 0), colspan=2)
ax3 = plt.subplot2grid((3, 3), (1, 2), rowspan=2)
ax4 = plt.subplot2grid((3, 3), (2, 0))
ax5 = plt.subplot2grid((3, 3), (2, 1))
```

Now if we want to plot data to any of these axes, we can direct our plotting commands to the appropriate axes, e.g.,

```
ax1.plot([1,2,3,4])
```

To construct a similar plot in figurefirst, we use Inkscape to create an svg layout document by drawing boxes where the five axes should lie, then tag these boxes with names, e.g., ax1, ax2, etc. To make the figure in Python, we then construct a *FigureLayout* object by passing the path to the layout document:

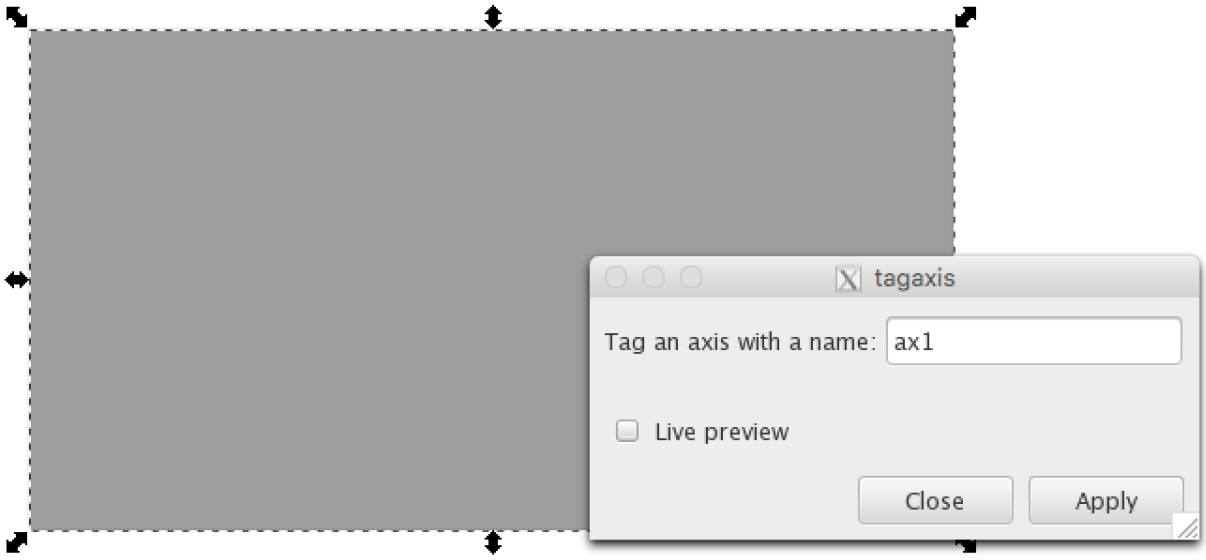
```
import figurefirst as fifi
layout = fifi.FigureLayout('/path/to/layout.svg', make_mplfigures=True)
```

figurefirst will construct the matplotlib figure and store the axes by their tag name in the 'axes' attribute of the layout object. For example, to plot on ax1:

```
layout.axes['ax1'].plot([1,2,3,4])
```

4.3 Designing a layout

So how do you design a layout? The easiest approach is to use Inkscape and take advantage of the figurefirst extensions to tag the axes with unique names. We create a new document specifying the height and width of the figure in the document properties menu. Next we use the rectangle tool to draw boxes where we want our axis to appear within the figure. Finally, we give these axes names using the tagaxis extension. The screenshot below shows the tagaxis dialog box being used to tag a box with the name 'ax1'.



4.4 Merging matplotlib plots back into the layout

Simply being able to use svg as a specification for axis placement is a useful feature on it's own, but figurefirst allows you to take things one step further and direct the output of your plotting code back into the svg layout document. This means that any svg vector art that you include in your layout document can now be used as overlays and underlays to your matplotlib figures. To save our new plots we simply call

```
layout.save('/path/to/output_file.svg')
```

By default figurefirst will create a new svg layer to place all the new matplotlib artwork. Note that merging your newly plotted figures with the layout in this way also provides a mechanism for fast iterative development of your figures.

If you save the plotted data back into the layout document (passing the original layout file as output_file), you can now make adjustments to the layout, add annotations, etc., now that you have a better idea of how the data lie within the figure. After making these changes, the plotting code can be re-run at any time to regenerate the plots without worrying about loosing these annotations.

By default, figurefirst will assume the layer called `"Layer 1"` is your template layer, and hide it. To disable this, set `hide_layers=[]` in the constructor. By preference, put non-template elements of your element on another layer.

CHAPTER 5

Examples

5.1 hello world

```
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt
plt.ion()
from figurefirst import FigureLayout

layout = FigureLayout('example_hello_world_layout.svg')
layout.make_mplfigures()

d = np.array([[144, 57], [138, 57], [138, 59], [141, 61], [141, 82], [138, 84], [138,␣
↪85], [142, 85], [147, 85], [147, 84], [144, 82], [144, 57], [144, 57], [155, 57],␣
↪[149, 57], [149, 59], [152, 61], [152, 82], [149, 84], [149, 85], [153, 85], [158,␣
↪85], [158, 84], [155, 82], [155, 57], [155, 57], [273, 57], [267, 57], [267, 59],␣
↪[270, 61], [270, 82], [267, 84], [267, 85], [271, 85], [276, 85], [276, 84], [273,␣
↪82], [273, 57], [273, 57], [295, 57], [289, 57], [289, 59], [292, 61], [292, 70],␣
↪[287, 67], [278, 76], [287, 85], [292, 83], [292, 85], [298, 85], [298, 84], [295,␣
↪81], [295, 57], [295, 57], [90, 57], [90, 59], [91, 59], [94, 61], [94, 82], [91,␣
↪84], [90, 84], [90, 85], [96, 85], [102, 85], [102, 84], [101, 84], [98, 82], [98,␣
↪71], [110, 71], [110, 82], [107, 84], [106, 84], [106, 85], [112, 85], [118, 85],␣
↪[118, 84], [117, 84], [113, 82], [113, 61], [117, 59], [118, 59], [118, 57], [112,␣
↪58], [106, 57], [106, 59], [107, 59], [110, 61], [110, 70], [98, 70], [98, 61],␣
↪[101, 59], [102, 59], [102, 57], [96, 58], [90, 57], [90, 57], [193, 57], [193, 59],
↪ [197, 60], [205, 85], [205, 86], [206, 85], [213, 65], [219, 85], [220, 86], [221,␣
↪85], [229, 61], [233, 59], [233, 57], [229, 58], [224, 57], [224, 59], [228, 61],␣
↪[227, 62], [221, 80], [215, 60], [215, 60], [218, 59], [218, 57], [213, 58], [208,␣
↪57], [208, 59], [211, 60], [212, 63], [207, 80], [200, 60], [200, 60], [203, 59],␣
↪[203, 57], [198, 58], [193, 57], [193, 57], [128, 67], [120, 76], [129, 85], [135,␣
↪80], [135, 80], [134, 80], [129, 84], [125, 82], [123, 76], [134, 76], [135, 75],␣
↪[128, 67], [128, 67], [169, 67], [160, 76], [169, 85], [178, 76], [169, 67], [169,␣
↪67], [240, 67], [231, 76], [240, 85], [249, 76], [240, 67], [240, 67], [257, 67],␣
↪[251, 68], [251, 69], [254, 71], [254, 82], [251, 84], [251, 85], [256, 85], [261,␣
↪85], [261, 84], [260, 84], [257, 82], [257, 75], [262, 68], [262, 68], [261, 70],␣
↪[263, 71], [265, 70], [262, 67], [257, 71], [257, 67], [257, 67], [128, 68], [133,␣
↪75], [123, 75], [128, 68], [128, 68], [169, 68], [173, 70], [174, 76], [173, 81],␣
↪[169, 84], [164, 82], [163, 76], [164, 70], [169, 68], [169, 68], [240, 68], [244,␣
↪70], [246, 76], [245, 81], [240, 84], [235, 82], [234, 76], [235, 70], [240, 68],␣
↪[240, 68], [287, 68], [292, 70], [292, 72], [292, 80], [292, 82], [287, 84], [283,␣
```

(continues on next page)

(continued from previous page)

```
ax = layout.axes['ax_name']['axis']

ax.plot(d[:,0], -d[:,1], lw=4)

layout.insert_figures('target_layer_name')

layout.write_svg('example_hello_world_output.svg')
```

5.2 grouped axes

```
#!/usr/bin/env python
from pylab import *
from figurefirst import FigureLayout,mpl_functions

#Group axes example
layout = FigureLayout('example_group_axes_layout.svg')
layout.make_mplfigures()
layout.insert_figures()
layout.write_svg('example_group_axes_output.svg')
close('all')
```

5.3 nested groups

```
#!/usr/bin/env python
import figurefirst as fifi
import matplotlib.pyplot as plt
import matplotlib

layout = fifi.FigureLayout('example_nested_groups_layout.svg')
layout.make_mplfigures()
#print layout.axes
#layout.axes_groups['fig2']['group3']['ax2'].plot([2,3,4])
for key,ax in layout.axes.items():
        #print key
        #print type(ax)
        ax.plot([1,3,2,4])

cdict = {'r1':0.3,'r2':0.1,'r3':0.9}
for key,value in cdict.items():
    hexi = matplotlib.colors.rgb2hex(plt.cm.viridis(value))
    layout.svgitems['svggroup'][key].style['fill'] = str(hexi)

layout.apply_svg_attrs()
layout.insert_figures()
layout.write_svg('example_nested_groups_output.svg')
```

5.4 axis methods

```
#!/usr/bin/env python
from pylab import *
from figurefirst import FigureLayout,mpl_functions

# You can decorate the svg <figurefirsrt:axis> tag with mpl.axis methods. For example␣
↪to call ax.axhspan(100,200,zorder=10,color ='r',alpha = 0.3) on the axis named␣
↪frequency.22H05.start use the following tag:
#
# ```
# <figurefirst:axis
#      figurefirst:name="frequency.22H05.start"
#      figurefirst:axhspan="100,200,zorder=10,color='r',alpha=.3"/> ```
#
# The layout.apply_mpl_methods function will then apply the methods passing the value␣
↪of the svg atribute as arguments to the mpl.axis method.

#Passing axis methods
import numpy as np
layout = FigureLayout('example_axis_methods_layout.svg')
layout.make_mplfigures()
layout.fig.set_facecolor('None')
for mplax in layout.axes.values():
    ax = mplax['axis']
    ax.plot(np.arange(30),np.random.rand(30),color = 'k')
    mpl_functions.adjust_spines(ax,'none',
                  spine_locations={},
                  smart_bounds=True,
                  xticks=None,
                  yticks=None,
                  linewidth=1)
    ax.patch.set_facecolor('None')
layout.apply_mpl_methods()
layout.insert_figures('mpl_panel_a')
layout.write_svg('example_axis_methods_output.svg')
close('all')
```

5.5 figure templating

```
#!/usr/bin/env python
import figurefirst

'''
To create a templated figure, draw a rectangle, and add a figurefirst:figure tag as␣
↪if you were creating another figure.
Then in addition to the figurefirst:name attribute, add a figurefirst:template␣
↪attribute,
whose value should correspond to the name of the figurefirst figure you want to use␣
↪as the template.
The template figure will be scaled to fill the box.
```

(continues on next page)

(continued from previous page)

```
'''

layout = figurefirst.FigureLayout('example_figure_templating_layout.svg' )
layout.make_mplfigures()
layout.append_figure_to_layer(layout.figures['group2'], 'mpl_layer_2')
layout.append_figure_to_layer(layout.figures['group3'], 'mpl_layer_3')
layout.write_svg('example_figure_templating_output.svg')
```

CHAPTER 6

API Reference

6.1 figurefirst package

6.1.1 Submodules

6.1.2 figurefirst.figurefirst_user_parameters module

6.1.3 figurefirst.mpl_functions module

figurefirst.mpl_functions.**add_mpl_patch**(*ax*, *patchname*, **args*, ***kwargs*)
wrapper for these two calls: patch = matplotlib.patches.Patch() ax.add_artist(patch)

figurefirst.mpl_functions.**adjust_spines**(*ax*, *spines*, *spine_locations={}*, *smart_bounds=True*, *xticks=None*, *yticks=None*, *linewidth=1*, *spine_location_offset=None*, *default_ticks=False*, *tick_length=5*, *color='black'*, *direction='in'*)
ax - matplotlib axis object spines - list of spines to include, e.g. ['left', 'bottom'] spine_locations - dict of spine locations, e.g. {'left': 5, 'bottom': 5}. Defaults to user_parameters. smart_bounds - if True, stop spine at first and last ticks xticks - list of locations for the xticks, e.g. [0, 5, 10] yticks - same as xticks, for y axis linewidth - width of spine and tick marks spine_location_offset - where to place spines. Overrides spine_locations if not None. default_ticks - if True, skip smart bounds and tick placement. NEcessary for loglog plots to work properly.

figurefirst.mpl_functions.**fix_mpl_svg**(*file_path*, *pattern='miterlimit:100000;'*, *subst='miterlimit:1;'*)
used to fix problematic outputs from the matplotlib svg generator, for example matplotlib creates exceptionaly large meterlimis

figurefirst.mpl_functions.**kill_all_labels**(*layout*)

figurefirst.mpl_functions.**kill_all_spines**(*layout*)

figurefirst.mpl_functions.**kill_labels**(*ax*)

figurefirst.mpl_functions.**kill_spines**(*ax*)

figurefirst.mpl_functions.**set_fontsize**(*fig*, *fontsize*)
For each text object of a figure fig, set the font size to fontsize fig can also be an axis object

figurefirst.mpl_functions.**set_spines**(*layout*)

6.1.4 figurefirst.svg_to_axes module

class figurefirst.svg_to_axes.**FFAxis**(*tagnode*, ***kwargs*)
Bases: *figurefirst.svg_to_axes.FFItem*

Stores the data requred for the creation of a matplotlib axes as specified by the layout

class figurefirst.svg_to_axes.**FFFigure**(*tagnode*, ***kwargs*)
Bases: *figurefirst.svg_to_axes.FFGroup*, object

Represents a matplotlib figure

class figurefirst.svg_to_axes.**FFGroup**(*tagnode*, ***kwargs*)
Bases: *figurefirst.svg_to_axes.FFItem*, object

used to collect groups objects that will be translated into matplotlib objects eg. axes and figures, x,y w and h is the collective x,y width and height of all the enclosed objects

class figurefirst.svg_to_axes.**FFItem**(*tagnode*, ***kwargs*)
Bases: dict, object

base class for figurefirst objects that will be converted to matplotlib objects eg. axes

class figurefirst.svg_to_axes.**FFSVGGroup**(*tagnode*, ***kwargs*)
Bases: *figurefirst.svg_to_axes.FFItem*, object

used to collect groups of svg items

class figurefirst.svg_to_axes.**FFSVGItem**(*tagnode*)
Bases: *figurefirst.svg_to_axes.FFItem*, object

base class for svg objects that figurefirst will manipulate in the native svg form eg. text and paths. These objects can be used to change the style of tagged graphics in the document

frmtstyle()

load_style()

class figurefirst.svg_to_axes.**FFSVGPath**(*tagnode*)
Bases: *figurefirst.svg_to_axes.FFSVGItem*, object

represents a svg path, currently figurefirst is unable to change the dimensions of the path, and the x, y, w, h attributes are meaningless placeholders

frmtstyle()

load_style()

class figurefirst.svg_to_axes.**FFSVGText**(*tagnode*)
Bases: *figurefirst.svg_to_axes.FFSVGItem*, object

represents a svg text object, you can change the text, font as well as styling x,y contain the orign of the text box, height and with are currently not implemented since that would require parsing the font info

load_style()

load_text()

class figurefirst.svg_to_axes.**FFTemplateTarget**(*tagnode*, ***kwargs*)
Bases: *figurefirst.svg_to_axes.FFFigure*, object

represtents the target of a template. The all the axes within the template figure will be scaled to fit within the template target box

class figurefirst.svg_to_axes.**FigureLayout**(*layout_filename*, *autogenlayers=True*, *make_mplfigures=False*, *dpi=300*, *hide_layers=('Layer 1',)*)
Bases: object

add_attribute_to_layer(*layer_name*, *attribute*, *value*)

append_figure_to_layer(*fig*, *fflayername*, *cleartarget=False*, *save_traceback=False*, *notes=None*)
inserts a figure object, fig, into an inkscape SVG layer, the layer fflayername should be taged with a figurefirst:targetlayer tag. if fflayername is not found then a targetlayer is generated so long as self.autogenlayers is set to True, the default. If cleartarget is set to True then the contents of the target layer will be removed, usefull for itterative figure design.

save_traceback - save the traceback stack to the figure's xml. This makes it easy to find out what function was
used to generate the figure. Also saves date modified

notes - string, which is added as an attribute to the xml of the layer. Helpful if you want to embed info into the
layers for future reference.

apply_mpl_methods()
apply valid mpl methods to figure

apply_svg_attrs(*svg_items_to_update='all'*)
applies attributes to svgitems eg. lw stroke ect... need to call this function before saving in order to propegate changes to svgitems

svg_items_to_update - (optional) - provide a list of the svgitem tags that you wish to have updated. Default is 'all', wl
which could reset svgitems whose attributes were not set since the last call to 'None'.

clear_fflayer(*fflayername*)
If inserting mpl figure into a mpl target layer that has stuff in it, e.g. old mpl data, use this function to clear the layer of any children except the figurefirst:targetlayer node.

create_new_targetlayer(*layer_name*)

from_userx(*x*, *dst*)
transfrom from user coords to dst coords

from_usery(*y*, *dst*)
same as userx but for y coords,not used in this version

get_figure_element_by_name(*name*)
finds the xmlnode with a given figurefirst:name tag

get_outputfile_layers()
returns dictionary of layers in teh output file

insert_figures(*fflayername='mpl_layer'*, *cleartarget=False*)
takes a reference to the matplotlib figure and saves the svg data into the target layer specified with the xml tag <figurefirst:targetlayer> this tag must have the attribute figurefirst:name = fflayername

load_pathspecs()
parses pathspec objects from the layout document

load_svgitems()
loads the svgitems from the layout document

make_breadcrumbs_for_axes()
Save layout.svg, data.svg, figure, axis information into each mpl axis

make_group_tree()
does the work of traversing the svg document and building the representation of figures, groups and axes, this needs to be done before self.makemplfigures can generate the matplotlib figurs

make_mplfigures(*hide=False*, *axes_order={}*)
generates matplotlib figures from the tree of parsed FFFigure and FFGroup, FFTemplatetargets

axes_order: {'figure1': [top_axis, next_axis, etc.], 'figure2': []}. Dictionary of the figures, that point to the list of axes, starting with the axis that should be on top first. Any axes not listed will end up on the bottom.

pass_xml(*gid*, *key*, *value*)
pass key, value pair xml pair to group with ID gid gid should contain prefix 'figurefirst:' to ensure uniqueness usage: layout.pass_xml(gid,'jessyink:effectIn', 'name:appear;order:1;length:800')

save(*filename*, *hidelayers=()*, *targetlayer=None*, *fix_meterlimt=True*)
convenience function, inserts layers and then calls wirte_svg to save

set_layer_visibility(*inkscape_label='Layer 1'*, *vis=True*, *gid=None*)
appled to the output svg usefull to hide the design layers on the newly created figures

to_svg_buffer(*fig*)
writes a matplotlib figure into a stringbuffer and returns the buffer

write_fifidata(*info*, **args*, ***kwargs*)
Write arbitrary data to the figurefirst data file info - list of strings, e.g. ['title', 'description 1', 'description 2'] args - data you want to store in the data file

write_svg(*output_filename*)
writes the current output_xml document to output_filename

class figurefirst.svg_to_axes.**LineSpec**(**args*, ***kwargs*)
Bases: *figurefirst.svg_to_axes.PathSpec*

mplkwargs()

class figurefirst.svg_to_axes.**PatchSpec**(**args*, ***kwargs*)
Bases: *figurefirst.svg_to_axes.PathSpec*

mplkwargs()

class figurefirst.svg_to_axes.**PathSpec**(**args*, ***kwargs*)
Bases: dict

load(*ptag*)

figurefirst.svg_to_axes.**extractTreeByType**(*node*, *searchType*)
walk a nested dictionary and pop the items of a certain type from the dictionary

figurefirst.svg_to_axes.**filterTreeByType**(*node*, *searchType*)

figurefirst.svg_to_axes.**flatten_dict**(*d*)
unrap a nested dict, d, returns a new dictionary with that have tuples as keys that specify the path through the original dictionary tree to the leaf. e.g. {'l1key': {'l2key1':item1,'l2key2':item2}} will become {('l1key','l2key1'):item1,('l1key','l2key2'):item2}

figurefirst.svg_to_axes.**flatten_list**(*container*)

figurefirst.svg_to_axes.**get_elements_by_attr**(*xml_node*, *attribute*, *value*)
helper function for xml trees when using minidom

figurefirst.svg_to_axes.**get_transforms**(*node*, *tlist=[]*)
get the list of transforms applied to a given node, walking up the svg tree to fill tlist. Pass an empyt list to call

figurefirst.svg_to_axes.**parse_transform**(*transform_str*)
convert transforms into transformation matrix

figurefirst.svg_to_axes.**repar**(*val*, *unit*)
reparse a value and unit into a string for writing into an svg

figurefirst.svg_to_axes.**tounit**(*in_unit*, *dst_unit*)
returns a float with the value of string s in the units of dst

figurefirst.svg_to_axes.**upar**(*unit_st*)
Parse unit_st into num (float), unit (string) pair

6.1.5 Module contents

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index

K

L

M

P

R

S

T

U

W