aafigure Documentation

Release 0.5

Chris Liechti

March 24, 2010

CONTENTS

MANUAL

1.1 Overview

The original idea was to parse ASCII art images, embedded in reST documents and output an image. This would mean that simple illustrations could be embedded as ASCII art in the reST source and still look nice when converted to e.g. HTML.

aafigure can be used to write documents that contain drawings in plain text documents and these drawings are converted to appropriate formats for e.g. HTML or PDF versions of the same document.

Since then aafigure also grew into a standalone application providing a command line tool for ASCII art to image conversion.

1.1.1 ASCII Art

The term "ASCII Art" describes a wide field.

- (small) drawings found in email signatures

- smilies :-)

- raster images (this was popular to print images on text only printers a *few* years ago)

- simple diagrams using lines, rectangles, arrows

aafigure aims to parse the last type of diagrams.

1.1.2 Other text to image tools

There are of course also a lot of other tools doing text to image conversions of some sort. One of the main differences is typically that other tools use a description language to generate images from rules. This is a major difference to aafigure which aims to convert good looking diagrams/images etc. in text files to better looking images as bitmap or vector graphics.

Graphviz Graphviz is a very popular tool that is excellent for displaying graphs and networks. It does this by reading a list of relations between nodes and it automatically finds the best way to place all the nodes in a visually appealing way.

This is quite different from aafigure and both have their strengths. Graphviz is very well suited to document state machines, class hierarchies and other graphs.

1.2 Installation

1.2.1 aafigure

This installs a package that can be used from python (`import aafigure`) and a command line script called `aafigure`.

The Python Imaging Library (PIL) needs to be installed when support for bitmap formats is desired and it will need ReportLab for PDF output.

To install the module for all users on the system, administrator rights (root) is required..

From source (tar.gz or checkout)

Unpack the archive, enter the `aafigure-x.y` directory and run:

```
python setup.py install
```

Setuptools/PyPI

Alternatively it can be installed from PyPy, either manually downloading the files and installing as described above or using:

```
easy_install -U aafigure
```

Packages

There are also packaged versions for some Linux distributions and Windows:

Ubuntu Add the repository to `/etc/apt/sources.list` as described on this page: https://launchpad.net/~aafigure-team/+archive/ppa

Then run `apt-get update` and `apt-get install aafigure`

Arch Linux "aafigure" (or "aafigure-bzr") are found in the category "unsupported".

Windows For users that have Python already installed, there is an installer for the extension on http://pypi.python.org/pypi/aafigure

1.2.2 Docutils plug-in

The docutils-aafigure extension depends on the aafigure package also requires `setuptools` (often packaged as `python-setuptools`) and Docutils itself (0.5 or newer) must be installed.

After that, the `aafigure` directive will be available.

1.2.3 Sphinx plug-in

sphinxcontrib-aafig is a plug-in similar to the Docutils plug-in, but it automatically selects the image format depending on the output format.

XXX elaborate

1.2.4 MoinMoin plug-in

MoinMoin is a popular Wiki engine. The plug-in allows to use aafigure drawings within wiki pages.

Copy the file `aafig.py` from `examples/moinmoin` to `wiki/data/plugin/parser` of the wiki. The aafigure module itself needs to be installed for the Python version that is used to run MoinMoin (see above for instructions).

Tested with MoinMoin 1.8.

See also: http://moinmo.in/ParserMarket/AaFigure

1.3 Usage

1.3.1 Command line tool

```
aafigure test.txt -t png -o test.png
```

The tool can also read from standard in and supports many options. Please look at the command's help (or man page):

```
aafigure --help
```

1.3.2 Within Docutils

A `aafigure` directive is provided that allows to insert images:

```
.. aafigure::

    DD o--->
```

```
./rst2html.py README.txt >README.html
```

This results in the `README.html` file and a `.svg` file (or the specified file type) for each `aafigure`.

The resulting `README.html` file can be viewed with a SVG capable browser. It has been tested with Firefox 1.5, 2.0 and 3.0.

1.3.3 Within Sphinx

In `conf.py` add:

```
extensions = ['sphinxcontrib.aafig']
```

This provides the `aafig` directive:

```
.. aafig::

    DD o--->
```

The output format is automatically chosen depending on the generated document format (e.g. HTML or PDF).

1.3.4 Within MoinMoin

ASCII Art figures can be inserted into a MoinMoin WikiText page the following way:

```
{{{#!aafig scale=1.5 foreground=#ff1010
DD o--->
}}}
```

The parser name is `aafig` and options are appended, separated with spaces. Options that require a value take that after a = without any whitespace between option and value. Supported options are:

- `scale=<float>`
- `aspect=<float>`
- `textual`
- `proportional`
- `linewidth=<float>`
- `foreground=#rrggbb`
- `fill=#rrggbb`

There is no `background` as the SVG backend ignores that. And it is not possible to pass generic options.

The images are generated and stored in MoinMoin's internal cache. So there is no mess with attached files on the page. Each change on an image generates a new cache entry so the cache may grow over time. However the files can be deleted with no problem as they can be rebuilt when the page is viewed again (the old files are not automatically deleted as they are still used when older revision of a page is displayed).

SHORT INTRODUCTION

2.1 Docutils & Sphinx integration

In a Sphinx document an image can be inserted like this:

```
.. aafig::

    -->
```

Which results in an image like this:

$\longrightarrow$

The same contents could also have been placed in a file and then be converted with the aafigure command line tool.

2.1.1 Docutils directive

The `aafigure` directive has the following options:

- `:scale:` `<float>` enlarge or shrink image

- `:line_width:` `<float>` change line with (svg only currently)

- `:format:` `<str>` choose backend/output format: 'svg', 'png', all bitmap formats that PIL supports can be used but only few make sense. Line drawings have a good compression and better quality when saved as PNG rather than a JPEG. The best quality will be achieved with SVG, tough not all browsers support this vector image format at this time.

- `:foreground:` `<str>` foreground color in the form `#rgb` or `#rrggbb`

- `:background:` `<str>` background color in the form `#rgb` or `#rrggbb` (*not* for SVG output)

- `:fill:` `<str>` fill color in the form `#rgb` or `#rrggbb`

- `:name:` `<str>` use this as filename instead of the automatic generated name

- `:aspect:` `<float>` change aspect ratio. Effectively it is the width of the image that is multiplied by this factor. The default setting `1` is useful when shapes must have the same look when drawn horizontally or vertically. However, `:aspect:` `0.5` looks more like the original ASCII and even smaller factors may be useful for timing diagrams and such. But there is a risk that text is cropped or is draw over an object beside it.

  The stretching is done before drawing arrows or circles, so that they are still good looking.

- `:proportional:` `<flag>` use a proportional font instead of a mono-spaced one.

2.1.2 Sphinx directive

It is called `aafig`. The same options as for the Docutils directive apply with the exception of `format`. That option is not supported as the format is automatically determined.

2.2 Lines

The – and | are normally used for lines. _ and ~ can also be used. They are slightly longer lines than the –. _ is drawn a bit lower and ~ a bit upper. = gives a thicker line. The later three line types can only be drawn horizontally.

It is also possible to draw diagonal lines. Their use is somewhat restricted though. Not all cases work as expected.
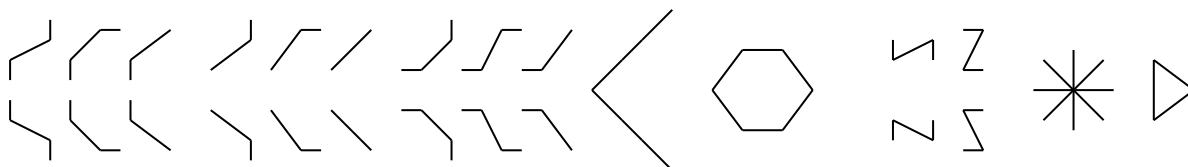
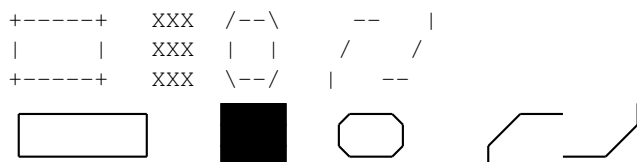And drawing longer diagonal lines with different angles looks ugly...
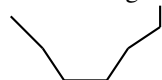

2.3 Arrows

Arrow styles are:
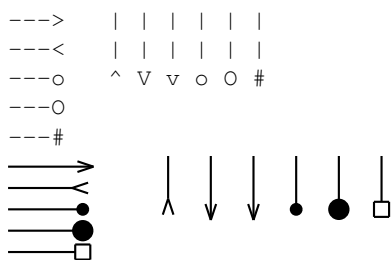
2.4 Boxes

Boxes are automatically draw when the edges are made with +, filled boxes are made with X (must be at least two units high or wide). It is also possible to make rounded edges in two ways:

2.5 Fills

Upper case characters generate shapes with borders, lower case without border. Fills must be at least two characters wide or high. (This reduces the chance that it is detected as Fill instead of a string)
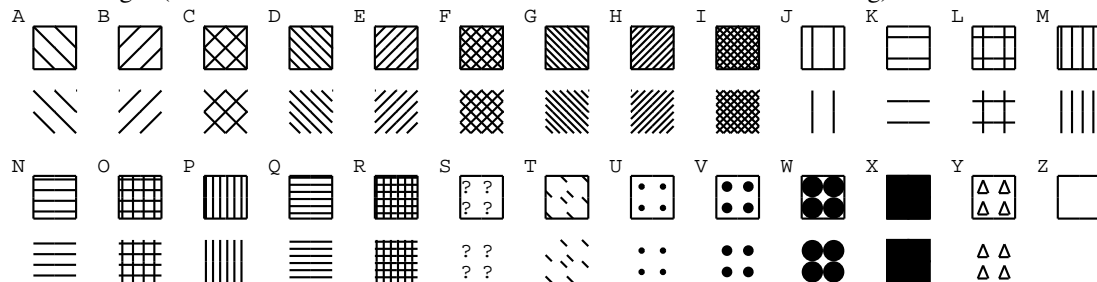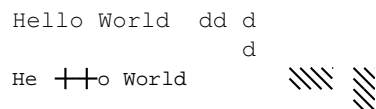
Complex shapes can be filled:

2.6 Text

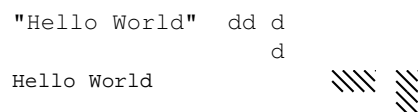The images may contain text too. There are different styles to enter text:

2.6.1 direct

By default are repeated characters detected as fill:

```
Hello World  dd d
                d
He  ++o World
```

2.6.2 quoted

Text between quotes has priority over any graphical meaning:

```
"Hello World"  dd d
                 d
Hello World
```

`"`, `'` and `` \` `` are all valid quotation marks. The quotes are not visible in the resulting image. This not only disables fills (see below), it also treats `-`, `|` etc. as text.

2.6.3 textual option

The `:textual:` option disables horizontal fill detection. Fills are only detected when they are vertically at least 2 characters high:
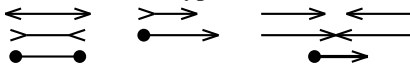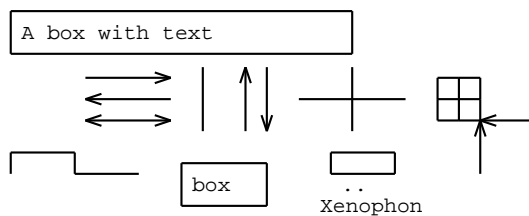
```
Hello World  dd d
                d
Hello World      dd
```
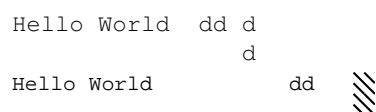
2.7 Other

```
*  {  }
```
● < >

EXAMPLES

3.1 Simple tests

Different arrow types:

Boxes and shapes:

```
A box with text
```

```
box
```

```
..
Xenophon
```
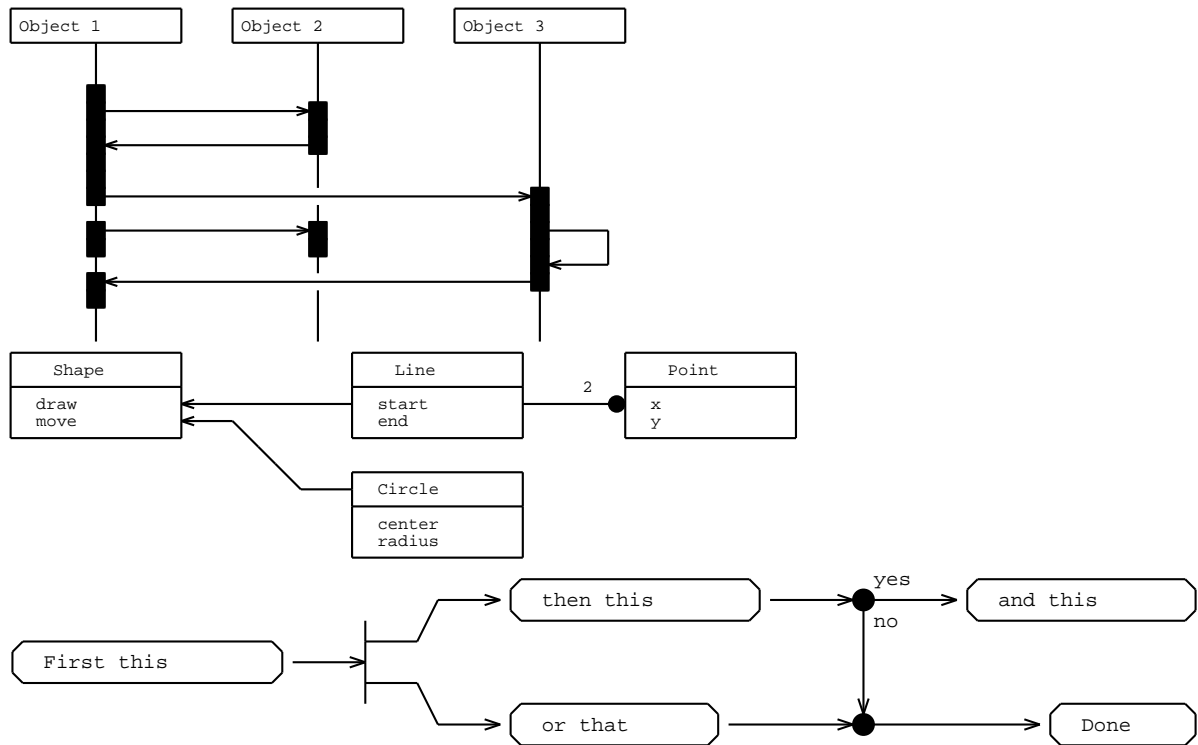
3.2 Flow chart
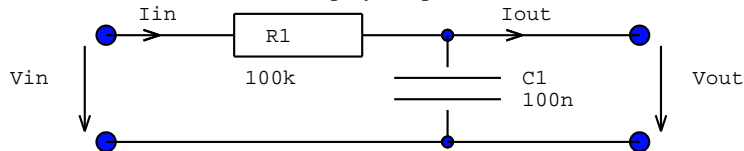
Start

Init

Process

more?

yes

no

End
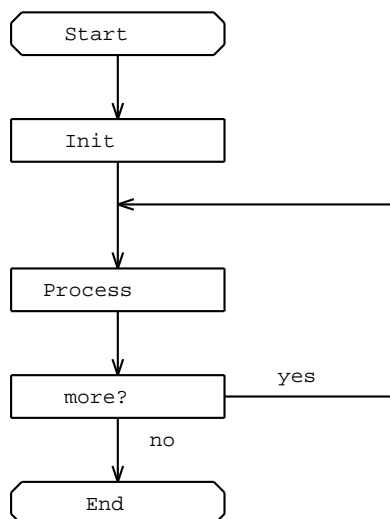
3.3 UML

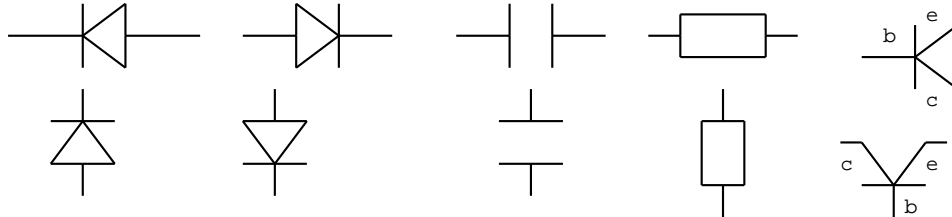No not really, yet. But you get the idea.

3.4 Electronics

It would be cool if it could display simple schematics.

- Capacitor not good, would prefer `--||--` -> symbol detection

- Diodes OK

- Caps not optimal. Too far apart in image, not very good recognisable in ASCII. Space cannot be removed as the two + signs would be connected otherwise. The schematic below uses an other style.

- Arrows in transistor symbols can not be drawn

Here is a complete circuit with different parts:
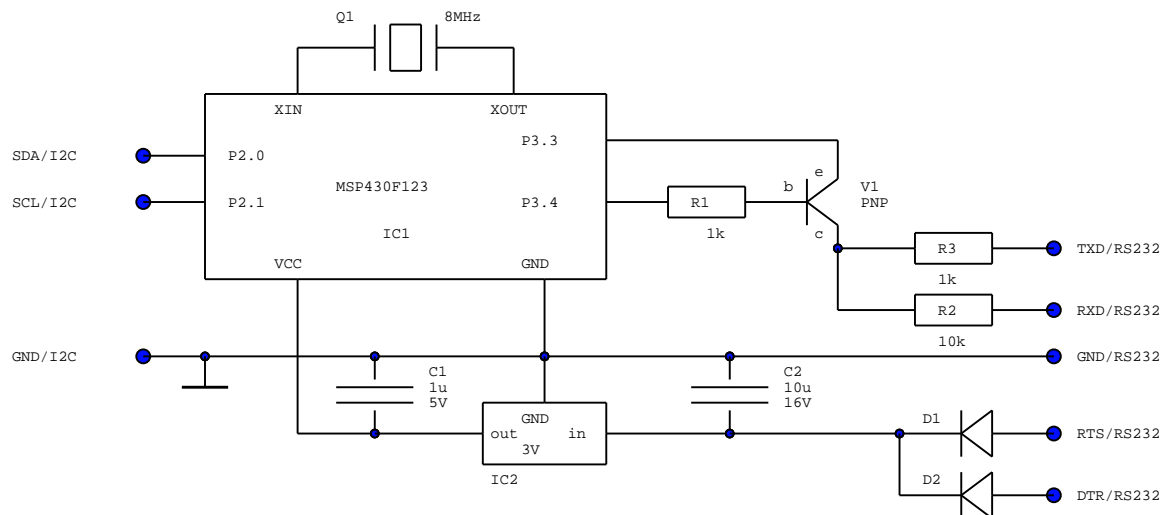
3.5 Timing diagrams
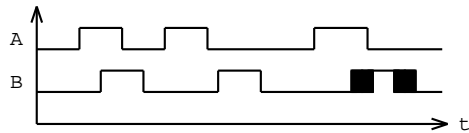
Here is one with descriptions:

3.6 Statistical diagrams

Benfords distribution of the sizes of files on my hard drive:

Just some bars:

3.7 Schedules

APPENDIX

4.1 API and Implementation Notes

4.1.1 External Interface

Most users of the module will use one of the following two functions. They provide a high level interface. They are also directly accessible as `aafigure.process` respectively `aafigure.render`.
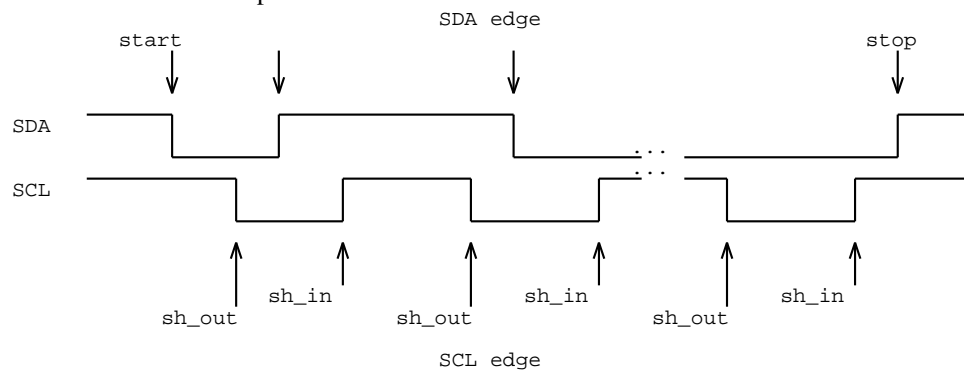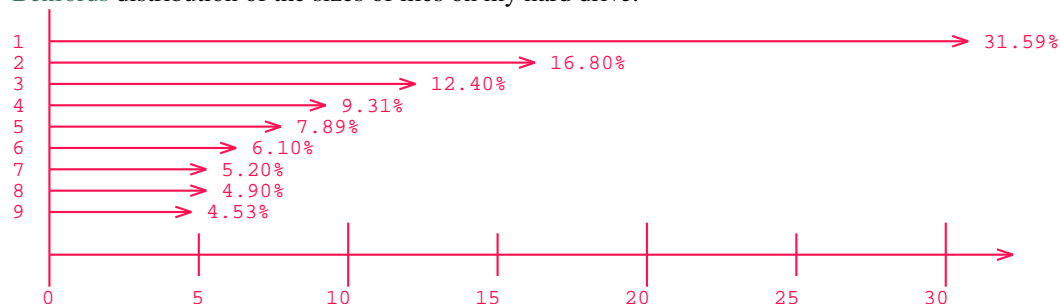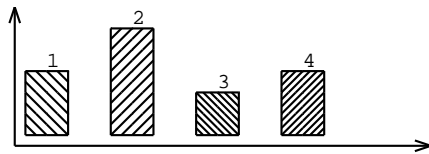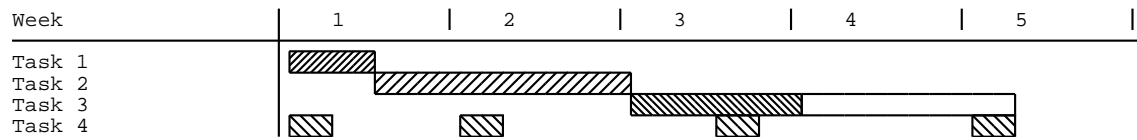
process(*input, visitor_class, options=None*)
> Parse input and render using the given visitor class.

> > **Parameters** • *input* – String or file like object with the image as text.

> > > • *visitor_class* – A class object, it will be used to render the resulting image.

> > > • *options* – A dictionary containing the settings. When `None` is given, defaults are used.

> > **Returns** instantiated `visitor_class` and the image has already been processed with the visitor.

> > **Exception** This function can raise an `UnsupportedFormatError` exception if the specified format is not supported.

render(*input, output=None, options=None*)
> Render an ASCII art figure to a file or file-like.

> > **Parameters** • *input* – If `input` is a basestring subclass (str or unicode), the text contained in `input` is rendered. If `input is a file-like object, the text to render is taken using ``input.read().`

> > > • *output* – If no `output` is specified, the resulting rendered image is returned as a string. If output is a basestring subclass, a file with the name of `output` contents is created and the rendered image is saved there. If `output` is a file-like object, `output.write()` is used to save the rendered image.

> > > • *options* – A dictionary containing the settings. When `None` is given, defaults are used.

> > **Returns** This function returns a tuple `(visitor, output)`, where `visitor` is visitor instance that rendered the image and `output` is the image as requested by the `output` parameter (a `str` if it was `None`, or a file-like object otherwise, which you should `close()` if needed).

> > **Exception** This function can raise an `UnsupportedFormatError` exception if the specified format is not supported.

The command line functionality is implemented in the `main` function.

main()
> implement an useful main for use as command line program

4.1.2 Internal Interface

The core functionality is implemented in the following class.

class `AsciiArtImage` (*text, aspect_ratio=1, textual=False, widechars='F, W'*)
> This class hold a ASCII art figure and has methods to parse it. The resulting list of shapes is also stored here.
>
> The image is parsed in 2 steps:
>
> > 1.horizontal string detection.
> >
> > 2.generic shape detection.
>
> Each character that is used in a shape or string is tagged. So that further searches don't include it again (e.g. text in a string touching a fill), respectively can use it correctly (e.g. join characters when two or more lines hit).
>
> > **`__init__`** (*text, aspect_ratio=1, textual=False, widechars='F, W'*)
> > > Take a ASCII art figure and store it, prepare for `recognize`
> >
> > **`recognize`** ()
> > > Try to convert ASCII art to vector graphics. The result is stored in `self.shapes`.

Images are built using the following shapes. Visitor classes must be able to process these types.

class `Arc` (*start, start_angle, end, end_angle, start_curve=True, end_curve=True*)
> A smooth arc between two points

class `Circle` (*center, radius*)
> Circle with center coordinates and radius.

class `Group` (*shapes=None*)
> A group of shapes

class `Label` (*position, text*)
> A text label at a position

class `Line` (*start, end, thick=False*)
> Line with starting and ending point. Both ends can have arrows

class `Point` (*x, y*)
> A single point. This class primary use is to represent coordinates for the other shapes.

class `Rectangle` (*p1, p2*)
> Rectangle with two edge coordinates.

`group` (*list_of_shapes*)
> return a group if the number of shapes is greater than one

`point` (*object*)
> return a Point instance. - if object is already a Point instance it's returned as is - complex numbers are converted to Points - a tuple with two elements (x,y)

4.1.3 Options

The `options` dictionary is used in a number of places. Valid keys (and their defaults) are:

Defining the output:

> **`file_like` <str>:** use the given file like object to write the output. The object needs to support a `.write(data)` method.
>
> **`format` <str>:** choose backend/output format: 'svg', 'pdf', 'png' and all bitmap formats that PIL supports can be used but only few make sense. Line drawings have a good compression and better quality when saved as PNG rather than a JPEG. The best quality will be achieved with SVG, tough not all browsers support this vector image format at this time (default: '`svg`').

Options influencing how an image is parsed:

textual <bool>: disables horizontal fill detection. Fills are only detected when they are vertically at least 2 characters high (default: `False`).

proportional <bool>: use a proportional font. Proportional fonts are general better looking than monospace fonts but they can mess the figure if you need them to look as similar as possible to the ASCII art (default: `False`).

Visual properties:

background <str>: background color in the form `#rgb` or `#rrggbb`, *not* for SVG output (default: `#000000`).

foreground <str>: foreground color in the form `#rgb` or `#rrggbb` (default: `#ffffff`).

fill <str>: fill color in the form `#rgb` or `#rrggbb` (default: same as `foreground` color).

line_width <float>: change line with, SVG only currently (default: `2.0`).

scale <float>: enlarge or shrink image (default: `1.0`).

aspect <float>: change aspect ratio. Effectively it is the width of the image that is multiplied by this factor. The default setting 1 is useful when shapes must have the same look when drawn horizontally or vertically. However, 0.5 looks more like the original ASCII and even smaller factors may be useful for timing diagrams and such. But there is a risk that text is cropped or is drawn over an object besides it.

The stretching is done before drawing arrows or circles, so that they are still good looking (default: `1.0`).

Miscellaneous options:

debug <bool>: for now, it only prints the original ASCII art figure text (default: `False`).

4.1.4 Visitors

A visitor that can be used to render the image must provide the following function (it is called by `process()`)

class **Visitor**()

> **visit_image**(*aa_image*)
> An `AsciiArtImage` instance is passed as parameter. The visiting function needs to implement a loop processing the `shapes` attribute.
>
> This function must take care of actually outputting the resulting image or it must provide the data in a form useful for the caller (`process()` returns the visitor so that the result can be read for example).

Example stub class:

```python
class Visitor:
    def visit_image(self, aa_image):
        self.visit_shapes(aa_image.shapes)

    def visit_shapes(self, shapes):
        for shape in shapes:
            shape_name = shape.__class__.__name__.lower()
            visitor_name = 'visit_%s' % shape_name
            if hasattr(self, visitor_name):
                getattr(self, visitor_name)(shape)
            else:
                sys.stderr.write("WARNING: don't know how to handle shape %r\n"
                    % shape)

    def visit_group(self, group):
```

```
        self.visit_shapes(group.shapes)

    # for actual output implement visitors for all the classes in
    # aafigure.shapes:

    def visit_line(self, lineobj):
        ...
    def visit_circle(self, circleobj):
        ...
    etc...
```

4.1.5 Source tree

The sources can be checked out using bazaar:

```
bzr lp:aafigure
```

Files in the `aafigure` package:

aafigure.py ASCII art parser. This is the main module.

shapes.py Defines a class hierachy for geometric shapes such as lines, circles etc.

error.py Define common exception classes.

aa.py ASCII art output backend. Intended for tests, not really useful for the end user.

pdf.py PDF output backend. Depends on reportlab.

pil.py Bitmap output backend. Using PIL, it can write PNG, JPEG and more formats.

svg.py SVG output backend.

Files in the `docutils` directory:

aafigure_directive.py Implements the `aafigure` Docutils directive that takes these ASCII art figures and generates a drawing.

The `aafigure` module contains code to parse ASCII art figures and create a list of of shapes. The different output modules can walk through a list of shapes and write image files.

4.1.6 TODO

- Symbol detection: scan for predefined shapes in the ASCII image and output them as symbol from a library

- Symbol libraries for UML, flowchart, electronic schematics, ...

- The way the image is embedded is a hack (inserting a tag trough a raw node...)

- Search for ways to bring in color. Ideas:

    - have an :option: to set color tags. Shapes that touch such a tag inherit it's color. The tag would be visible in the ASCII source tough:

      ```
      .. aafig::
          :colortag: 1:red, 2:blue

          1--->  --->2
      ```

    - :color: x,y,color but counting coordinates is no so fun

    drawback: both are complex to implement, searching for shapes that belong together. It's also not always wanted that e.g. when a line touches a box, both have the same color

- aafigure probably needs arguments like `font-family`, ...

- Punctuation not included in strings (now a bit improved but if it has a graphical meaning , then that is chooses, even if it makes no sense), underlines in strings are tricky to detect...

- Dotted lines? `...` e.g. for `--- ... ---` insert a dashed line instead of 3 textual dots. Vertical dashed lines should also work with `:`.

- Group shapes that belong to an object, so that it's easier to import and change the graphics in a vector drawing program. [partly done]

- Path optimizer, it happens that many small lines are output where a long line could be used.

4.2 Authors and Contact

- Chris Liechti: original author
- Leandro Lucarella: provided many patches

The project page is at https://launchpad.net/aafigure It should be used to report bugs and feature requests.

4.3 License

Copyright (c) 2006-2009 aafigure-team All rights reserved.

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

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- 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.

- Neither the name of the aafigure-team nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE AAFIGURE-TEAM ''AS IS" AND ANY EXPRESS OR IM-PLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MER-CHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AAFIGURE-TEAM BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EX-EMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUP-TION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

A

INDEX