

---

# aafigure



## aafigure Documentation

*Release 0.5*

**Chris Liechti**

**Jun 06, 2017**



---

## Contents

---

<b>1</b>	<b>Manual</b>	<b>1</b>
1.1	Overview . . . . .	1
1.2	Installation . . . . .	2
1.3	Usage . . . . .	2
<b>2</b>	<b>Short introduction</b>	<b>3</b>
2.1	Lines . . . . .	3
2.2	Arrows . . . . .	4
2.3	Boxes . . . . .	4
2.4	Fills . . . . .	4
2.5	Text . . . . .	4
2.6	Other . . . . .	5
<b>3</b>	<b>Examples</b>	<b>7</b>
3.1	Simple tests . . . . .	7
3.2	Flow chart . . . . .	8
3.3	UML . . . . .	8
3.4	Electronics . . . . .	8
3.5	Timing diagrams . . . . .	9
3.6	Statistical diagrams . . . . .	10
3.7	Schedules . . . . .	10
<b>4</b>	<b>Integrations</b>	<b>11</b>
4.1	Sphinx . . . . .	11
4.2	Docutils . . . . .	13
4.3	MoinMoin plug-in . . . . .	13
<b>5</b>	<b>Appendix</b>	<b>15</b>
5.1	API and Implementation Notes . . . . .	15
5.2	3rd party integration of aafigure . . . . .	19
5.3	Authors and Contact . . . . .	19
5.4	License . . . . .	19
<b>6</b>	<b>Indices and tables</b>	<b>21</b>
	<b>Python Module Index</b>	<b>23</b>



## 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.

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

## 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

afigure aims to parse the last type of diagrams.

## 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 afigure which aims to convert good looking diagrams/images etc. in text files to better looking images as bitmap or vector graphics. Here are some examples (by no means a complete list):

**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.

**Mscgen** A tool that is specialized for sequence diagrams (used to describe software, UML).

**ditaa** Convert diagrams to images.

## Installation

### aafigure

```
pip install 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.

### Requirements

- **reportlab** (for LaTeX/PDF output)
- **PIL** or **Pillow** (for any image format other than SVG or PDF)

## Usage

### 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
```

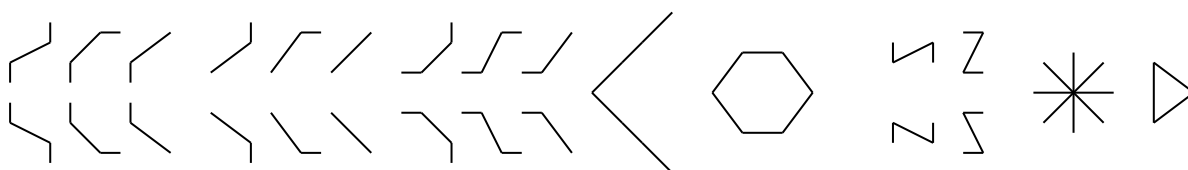
## Short introduction

### Lines

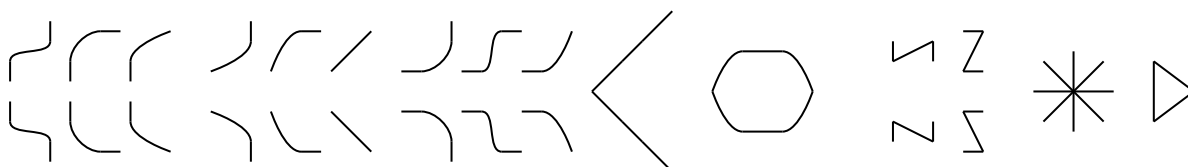
The `-` and `|` are normally used for lines. `_` and `~` can also be used. They are slightly longer lines than the `-`. `_` is drawn a bit lower and `~` a bit upper. `=` gives a thicker line. The later three line types can only be drawn horizontally.



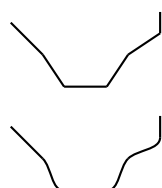
It is also possible to draw diagonal lines. Their use is somewhat restricted though. Not all cases work as expected.



With rounded flag:

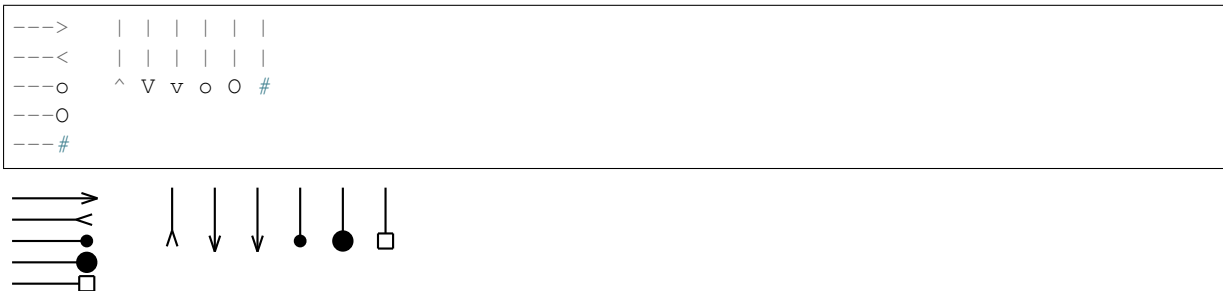


And drawing longer diagonal lines with different angles looks ugly...



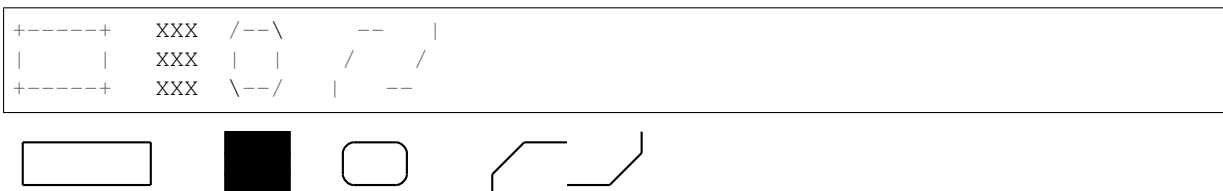
## Arrows

Arrow styles are:



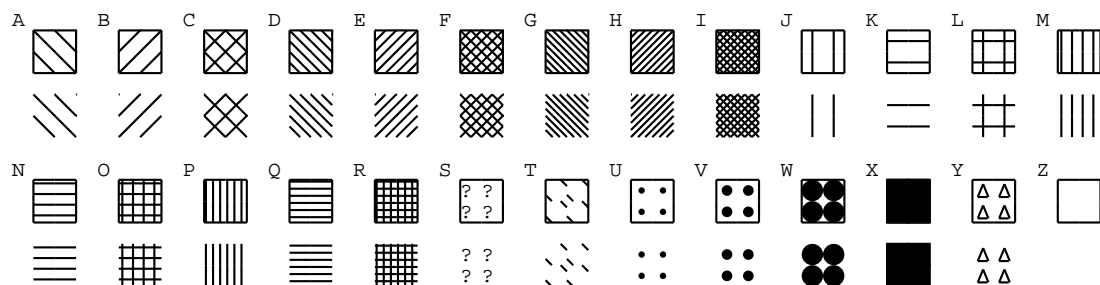
## Boxes

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:



## 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:



## Text


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

### direct

By default are repeated characters detected as fill:



```
Hello World  dd d
                d
```

He ++o World      

## 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.

## 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 

## Other

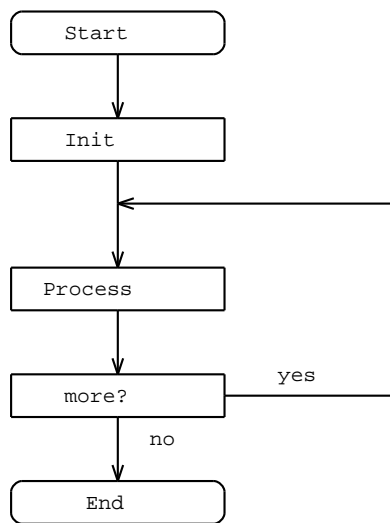
```
* { }
```

● < >



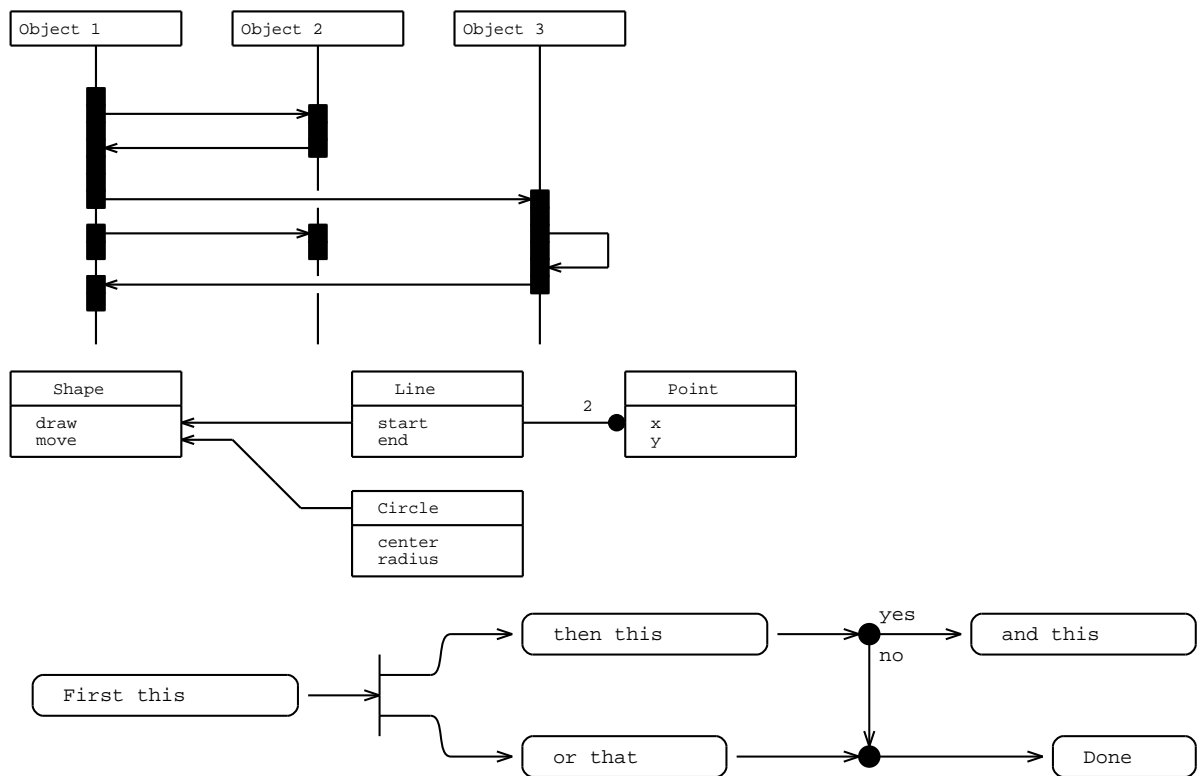


## Flow chart



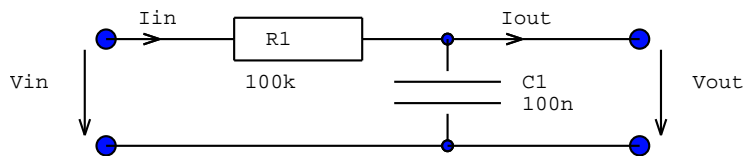
## UML

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

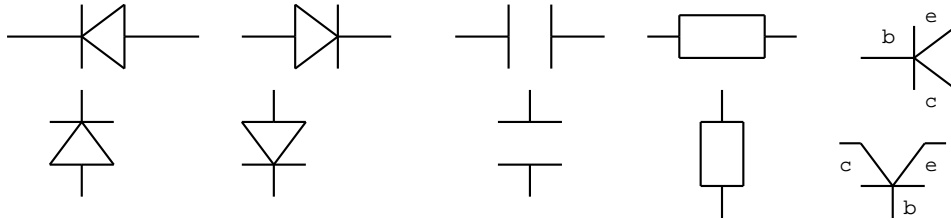


## Electronics

It would be cool if it could display simple schematics.

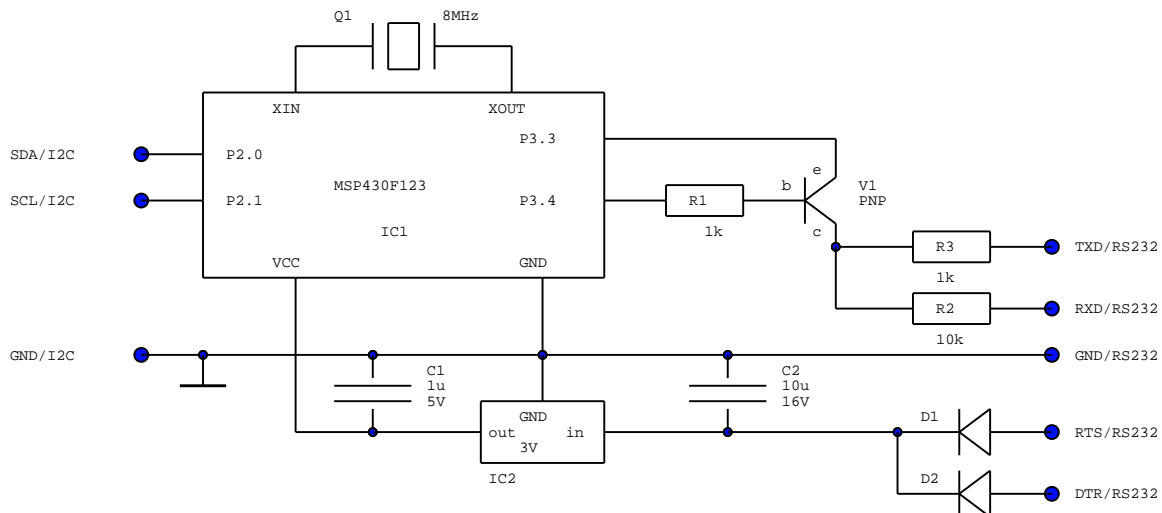


- Capacitor not good, would prefer -- | | -- -> symbol detection

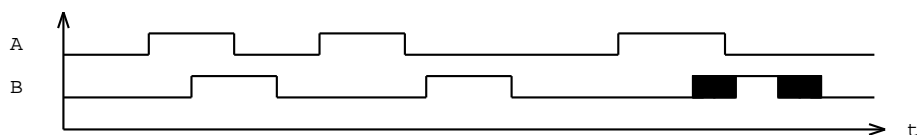


- Diodes OK
- Caps not optimal. Too far apart in image, not very good recognisable in ASCII. Space cannot be removed as the two + signs would be connected otherwise. The schematic below uses an other style.
- Arrows in transistor symbols can not be drawn

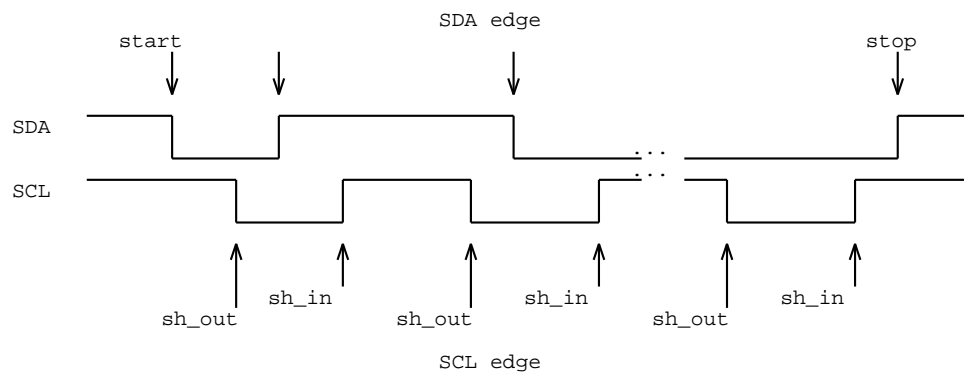
Here is a complete circuit with different parts:



## Timing diagrams

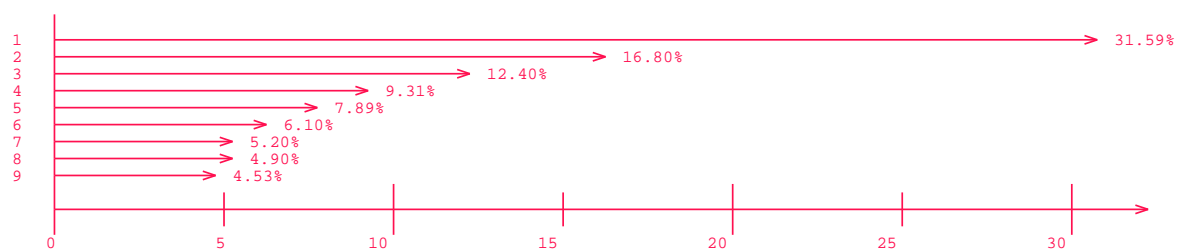


Here is one with descriptions:

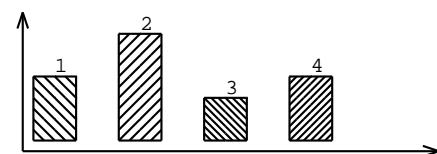


## Statistical diagrams

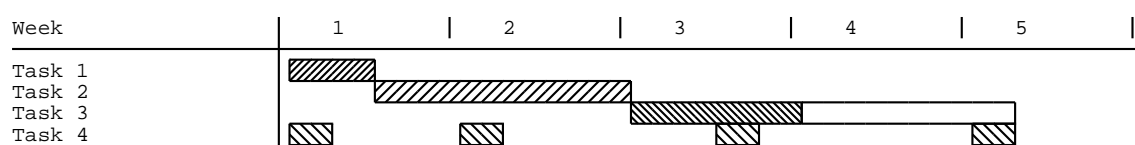
Benfords distribution of the sizes of files on my hard drive:



Just some bars:



## Schedules



## Sphinx

This extension adds the `aafig` directive that automatically selects the image format to use according to the [Sphinx](#) writer used to generate the documentation.

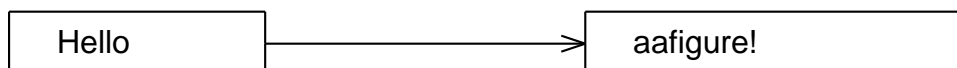
### Quick Example

This source:

```
.. aafig::
    :aspect: 60
    :scale: 150
    :proportional:
    :textual:

+-----+          +-----+
| Hello +----->+ aafigure! |
+-----+          +-----+
```

is rendered as:



### Enabling the extension in Sphinx

Just add `aafigure.sphinxext` to the list of extensions in the `conf.py` file. For example:

```
extensions = ['aafigure.sphinxext']
```

### Options

The `aafig` directive has the following options:

- `:scale:` `<int>` enlarge or shrink image
- `:line_width:` `<float>` change line width (SVG only currently)
- `: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`
- `:aspect:` `<int>` change aspect ratio. Effectively it is the width of the image that is multiplied by this percentage. The default setting 1 is useful when shapes must have the same look when drawn horizontally or vertically. However, `:aspect: 50` 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.

- `:proportional:` use a proportional font instead of a mono-spaced
- `:textual:` prefer to detect text instead of fills
- `:rounded:` use arcs instead of straight lines for many diagonals
- `:scale:` and `:aspect:` options are specified using percentages (without the `%` sign), to match the `reStructuredText` image directive.

## Configuration

A few configuration options are added (all optional, of course ;) to [Sphinx](#) so you can set them in the `conf.py` file:

**aafig\_format** `<dict>`: image format used for the different builders. All `latex`, `html` and `text` builder are supported, and it should be trivial to add support for other builders if they correctly handle images (and if afigure can render an image format suitable for that builder) by just adding the correct format mapping here.

A special format `None` is supported, which means not to use afigure to render the image, just show the raw ASCII art as is in the resulting document (using a literal block). This is almost only useful for the text builder.

You can specify the format - builder mapping using a dict. For example:

```
aafig_format = dict(latex='pdf', html='svg', text=None)
```

These are the actual defaults.

**aafig\_default\_options** `<dict>`: default afigure options. These options are used by default unless they are overridden explicitly in the `aafig` directive. The default afigure options are used if this is not specified. You can provide partial defaults, for example:

```
aafig_default_options = dict(scale=150, aspect=50, proportional=True)
```

Note that in this case the `aspect` and `scale` options are specified as floats, as originally done by afigure.

## TODO

- Add color validation for `fill`, `background` and `foreground` options.
- Add `aa` role for easily embed small images (like arrows).

## History

This extension was once shipped separately: [sphinxcontrib-aafig website](#).



## Docutils

The docutils directive is provided in *aafigure/docutils*.

### 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, though 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.

### 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.

## 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>

## Usage

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`
- `textual_strict`
- `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).

## API and Implementation Notes

### 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`.

`aafigure.aafigure.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.

`aafigure.aafigure.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.

```
aafigure.aafigure.main()
    implement an useful main for use as command line program
```

## Internal Interface

The core functionality is implemented in the following class.

```
class aafigure.aafigure.AsciiArtImage(text, options=None)
```

This class holds 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, options=None)
```

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 aafigure.shapes.Arc(start, start_angle, end, end_angle, start_curve=True,
                          end_curve=True)
```

A smooth arc between two points

```
class aafigure.shapes.Circle(center, radius)
```

Circle with center coordinates and radius.

```
class aafigure.shapes.Group(shapes=None)
```

A group of shapes

```
class aafigure.shapes.Label(position, text)
```

A text label at a position

```
class aafigure.shapes.Line(start, end, thick=False)
```

Line with starting and ending point. Both ends can have arrows

```
class aafigure.shapes.Point(x, y)
```

A single point. This class primary use is to represent coordinates for the other shapes.

```
class aafigure.shapes.Rectangle(p1, p2)
```

Rectangle with two edge coordinates.

```
aafigure.shapes.group(list_of_shapes)
```

return a group if the number of shapes is greater than one

```
aafigure.shapes.point(obj)
```

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)

## 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, though 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`).

**textual\_strict <bool>:** Disables fill detection completely. (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`).

## Visitors

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

```
class your.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:

```
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
    # afigure.shapes:

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

## Source tree

The sources can be checked out using `bazaar`:

```
bzr lp:afigure
```

Files in the `afigure` package:

**afigure.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:

**afigure\_directive.py** Implements the `afigure` Docutils directive that takes these ASCII art figures and generates a drawing.

The `afigure` 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.

## 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.

## 3rd party integration of aafigure

There are also other projects that integrate aafigure. The following items are maintained by other developers.

### MediaWiki Plug-in

MediaWiki is a popular implementation of a WikiWikiWeb which is also used for Wikipedia. A plug-in can be found here: <http://www.mediawiki.org/wiki/Extension:Aafigure>

### AsciiDoc Plug-in

AsciiDoc is a plain text documentation format that can be converted into several formats such as HTML or PDF. A plug-in to use aafigure drawings in such documents can be found here: <http://code.google.com/p/asciidoc-aafigure-filter/>

## 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.

## License

Copyright (c) 2006-2017 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 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AAFigure-TEAM BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) 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.



## CHAPTER 6

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



### a

`aafigure.aafigure`, [15](#)  
`aafigure.shapes`, [16](#)



## Symbols

`__init__()` (afigure.afigure.AsciiArtImage method),  
16

## A

afigure.afigure (module), 15

afigure.shapes (module), 16

Arc (class in afigure.shapes), 16

AsciiArtImage (class in afigure.afigure), 16

## C

Circle (class in afigure.shapes), 16

## G

Group (class in afigure.shapes), 16

group() (in module afigure.shapes), 16

## L

Label (class in afigure.shapes), 16

Line (class in afigure.shapes), 16

## M

main() (in module afigure.afigure), 16

## P

Point (class in afigure.shapes), 16

point() (in module afigure.shapes), 16

process() (in module afigure.afigure), 15

## R

recognize() (afigure.afigure.AsciiArtImage method),  
16

Rectangle (class in afigure.shapes), 16

render() (in module afigure.afigure), 15

## V

visit\_image() (your.Visitor method), 17

Visitor (class in your), 17