autobasedoc Documentation

Release 1.1.10

Johannes Eckstein, Oliver Braun

Mar 09, 2022

Contents:

CHAPTER 1

Install

1.1 Simple

If you want it simple, go to a console an type:

```
pip install autobasedoc
```

1.2 With virtualenv

Otherwise it may also be recommended to install it into a virtual environment. To that end type in a console:

```
virtualenv venv
source ./venv/bin/activate
pip install autobasedoc
```

1.3 Using Miniconda/Anaconda

I prefer miniconda but the package does not exist for conda, yet:

```
conda create -n abd-env

source activate abd-env

pip install autobasedoc
```

1.4 Python Compatibility

The module is compatible with Python 3.4+, currently Python 3.7.3, we do not support older Python 2.7+ anymore.

1.5 Platforms

The module is platform independent. Up to now it is tested on linux (ubuntu). Other platforms will follow (at least Windows 10 should work).

CHAPTER 2

Basic Usage

Short

- There are actually two modules: *autoplot* and *autorpt*
- The principle usage is shown in the tests and the example

2.1 Reportlab-Suite

To work with the reportlab-toolbox you should import the following:

```
from autobasedoc import autorpt as ar
```

A usual working example for creating just a simple (empty) pdf file with a title-page and a table of contents would be:

```
import os
outname = os.path.join(os.path.dirname(__file__), "MinimalExample.pdf")
doc = ar.AutoDocTemplate(outname,onFirstPage=ar.onFirstPage,onLaterPages=ar.
↪onLaterPages,onLaterSPages=ar.onLaterPages,
                        leftMargin=0.5*ar.cm, rightMargin=0.5*ar.cm, topMargin=0.5*ar.
↪cm, bottomMargin=0.5*ar.cm)

#  you always work with your styles object
styles = ar.Styles()
styles.registerStyles()

#  the container for the contents, also commonly called the story (contains reportlab␣
↪flowables)
content = []
#add title
para = ar.Paragraph(u"Minimal Example Title", styles.title)
```

(continues on next page)

(continued from previous page)

```
content.append(para)
content.append(ar.PageBreak())

#  create table of contents. Override the level styles (optional)
toc = ar.doTabelOfContents()
content.append(ar.Paragraph(u"Table Of Contents", styles.h1))
content.append(toc)

#  always call multi build at the end
doc.multiBuild(content)
```

2.2 ttf-Fonts

To work with ttf-fonts and have the same font inside your matplotlib images and reportlab.platypus you have to use the following code block:

```
import os  #  if you haven't, yet.
# we assume you have a fonts path
# (if you haven't you can use the calibri font we added to the module) ar.__font_dir__
# the ap (autoplot) module provides helpful stuff for combining reportlab with␣
↪matplotlib
from autobasedoc import ap

#  here should be your path to the fonts (you can also use system fonts)

ar.setTtfFonts(
    'Calibri',
    os.path.realpath(ar.__font_dir__),
    normal=('Calibri', 'calibri.ttf'),
    bold=('CalibriBd', 'calibrib.ttf'),
    italic=('CalibriIt', 'calibrii.ttf'),
    bold_italic=('CalibriBdIt', 'calibriz.ttf'))
```

But you also want to have your fonts in sync with matplotlib, that's why you additionally have to:

```
fpath = os.path.join(ar.__font_dir__, 'calibri.ttf')
font = ap.ft2font.FT2Font(fpath)
ap.fontprop = ap.ttfFontProperty(font)

fontprop = ap.fm.FontProperties(
    family='sans-serif',
    fname=ap.fontprop.fname,
    size=None,
    stretch=ap.fontprop.stretch,
    style=ap.fontprop.style,
    variant=ap.fontprop.variant,
    weight=ap.fontprop.weight)

fontsize = 10
ap.matplotlib.rcParams.update({
    'font.size': fontsize,
    'font.family': 'sans-serif'
    })
```

You might then additionally wan't to use the same colors, that reportlab uses:

```
from cycler import cycler

plotColorDict = dict(
    royalblue='#4169E1',
    tomato='#FF6347',
    gold='#FFD700',
    mediumturquoise='#48D1CC',
    mediumorchid='#BA55D3',
    yellowgreen='#9ACD32',
    burlywood='#DEB887',
    darkslategray='#2F4F4F',
    orange='#FFA500',
    silver='#C0C0C0')

plotColorNames = list(plotColorDict.keys())
plotColors = list(plotColorDict.values())

ap.plt.rc('axes', prop_cycle=(cycler('color', plotColors)))
```

CHAPTER 3

Examples

An example should always describe the full workflow, here a very basic example on the usage:

3.1 Prerequisites

There are some preparations to make before you can actually add content to the document.

```
import os
from cycler import cycler
from autobasedoc import autorpt as ar
from autobasedoc import autoplot as ap

ar.setTtfFonts(
    'Calibri',
    os.path.realpath(ar.__font_dir__),
    normal=('Calibri', 'calibri.ttf'),
    bold=('CalibriBd', 'calibrib.ttf'),
    italic=('CalibriIt', 'calibrii.ttf'),
    bold_italic=('CalibriBdIt', 'calibriz.ttf'))

plotColorDict = dict(
    royalblue='#4169E1',
    tomato='#FF6347',
    gold='#FFD700',
    mediumturquoise='#48D1CC',
    mediumorchid='#BA55D3',
    yellowgreen='#9ACD32',
    burlywood='#DEB887',
    darkslategray='#2F4F4F',
    orange='#FFA500',
    silver='#C0C0C0')

plotColorNames = list(plotColorDict.keys())
```

(continues on next page)

(continued from previous page)

```
plotColors = list(plotColorDict.values())

ap.plt.rc('axes', prop_cycle=(cycler('color', plotColors)))

fpath = os.path.join(ar.__font_dir__, 'calibri.ttf')
font = ap.ft2font.FT2Font(fpath)
ap.fontprop = ap.ttfFontProperty(font)

fontprop = ap.fm.FontProperties(
    family='sans-serif',
    fname=ap.fontprop.fname,
    size=None,
    stretch=ap.fontprop.stretch,
    style=ap.fontprop.style,
    variant=ap.fontprop.variant,
    weight=ap.fontprop.weight)

fontsize = 10
ap.matplotlib.rcParams.update({
    'font.size': fontsize,
    'font.family': 'sans-serif'
})

styles = ar.Styles()
styles.registerStyles()

outname = os.path.join(os.path.dirname(__file__), "MinimalExample.pdf")
# from django.http import HttpResponse
# outname = HttpResponse(mimetype='application/pdf')
# outname['Content-Disposition'] = 'attachment; filename=somefilename.pdf'

doc = ar.AutoDocTemplate(outname,onFirstPage=ar.onFirstPage,onLaterPages=ar.
↪onLaterPages,onLaterSPages=ar.onLaterPages,
                         leftMargin=0.5*ar.cm, rightMargin=0.5*ar.cm, topMargin=0.5*ar.
↪cm, bottomMargin=0.5*ar.cm)

content = []

#add title
para = ar.Paragraph(u"Minimal Example Title", styles.title)
content.append(para)
content.append(ar.PageBreak())

# Create Table Of Contents. Override the level styles (optional)
# and add the object to the story
toc = ar.doTabelOfContents()
content.append(ar.Paragraph(u"Inhaltsverzeichnis", styles.h1))
content.append(toc)
```

3.2 A simple Table

Adding a simple Table using just the normal way to do this with reportlab.

```
content.append(ar.PageBreak())
ar.addHeading("A Table", styles.h1, content)

para = ar.Paragraph(u"My Text that I can write here or take it from␣
↪somewhere like shown in the next paragraph.", styles.normal)
content.append(para)

content.append(ar.Paragraph("Table is here.",styles.caption))

data = [(1,2,3,4), (5,6,6,8)]

content.append(ar.Table(data, style=None, spaceBefore=10))
```

3.3 A simple Matplotlib Plot

You can use a decorator *@ap.autoPdfImg* from the autoplot module to turn a simple plot function into a reportlab flowable.

```
content.append(ar.PageBreak())
ar.addHeading("A simple Image", styles.h1, content)
content.append(ar.Paragraph("Pictures are to be placed here.",styles.normal))

title = "My simple plot"

@ap.autoPdfImg
def my_plot1(canvaswidth=5): #[inch]
    fig, ax = ap.plt.subplots(figsize=(canvaswidth,canvaswidth))
    fig.suptitle(title, fontproperties=fontprop)
    x=[1,2,3,4,5,6,7,8]
    y=[1,6,8,3,9,3,4,2]
    ax.plot(x,y,label="legendlabel")
    nrow, ncol = 1, 1
    handles, labels = ax.get_legend_handles_labels()

    leg_fig = ap.plt.figure(figsize=(canvaswidth, 0.2*nrow))

    ax.legend(handles, labels, #labels = tuple(bar_names)
            ncol=ncol, mode=None,
            borderaxespad=0.,
            loc='best',        # the location of the legend handles
            handleheight=None,   # the height of the legend handles
            #fontsize=9,         # prop beats fontsize
            markerscale=None,
            #frameon=False,
            prop=fontprop,
            fancybox=True
            )

    return fig

content.append(my_plot1())
para = ar.Paragraph(" ".join((u"Fig.",str(doc.figcounter()),title)), styles.
↪caption)
content.append(para)
```

(continues on next page)

(continued from previous page)

3.4 A simple Matplotlib Plot and a Legend

If you want to have a separate Legend that you can separately add to your document you can use the decorator *@ap.autoPdfImage*.

```
content.append(ar.PageBreak())
ar.addHeading("An Image with Legend", styles.h1, content)
content.append(ar.Paragraph("Pictures are to be placed here.",styles.normal))

title = "My plot with a separate legend"

@ap.autoPdfImage
def my_plot2(canvaswidth=5): #[inch]
    fig, ax = ap.plt.subplots(figsize=(canvaswidth,canvaswidth))
    fig.suptitle(title,fontproperties=fontprop)
    x=[1,2,3,4,5,6,7,8]
    y=[1,6,8,3,9,3,4,2]
    ax.plot(x,y,label="legendlabel")
    nrow, ncol = 1, 1
    handles, labels = ax.get_legend_handles_labels()

    leg_fig = ap.plt.figure(figsize=(canvaswidth, 0.2*nrow))

    leg = leg_fig.legend(handles, labels, #labels = tuple(bar_names)
            ncol=ncol,
            mode=None,
            borderaxespad=0.,
            loc='center',         # the location of the legend handles
            handleheight=None,    # the height of the legend handles
            #fontsize=9,          # prop beats fontsize
            markerscale=None,
            frameon=False,
            prop=fontprop
            #fancybox=True,
            )

    return fig,leg_fig,leg

img, leg = my_plot2()

content.append(leg)
content.append(img)
para = ar.Paragraph(" ".join((u"Fig.",str(doc.figcounter()),title)), styles.
↪caption)
content.append(para)
```

3.5 Building the contents

Finally build the whole document.

```
doc.multiBuild(content)
```

3.5.1 autoplot

Created on Wed Sep 16 11:11:12 2015

`autobasedoc.autoplot.`**`autoPdfImage`** (*func*)

decorator for the autoplot module

returns two PdfImage objects if wrapped plt-function obeys the principle demonstated in following minimal example:

```
@autoPdfImage
def my_plot(canvaswidth=5): #[inch]
    fig, ax = ap.plt.subplots(figsize=(canvaswidth,canvaswidth))
    fig.suptitle("My Plot",fontproperties=fontprop)
    x=[1,2,3,4,5,6,7,8]
    y=[1,6,8,3,9,3,4,2]
    ax.plot(x,y,label="legendlabel")
    nrow,ncol=1,1
    handles, labels = ax.get_legend_handles_labels()

    leg_fig = ap.plt.figure(figsize=(canvaswidth, 0.2*nrow))

    leg = leg_fig.legend(handles, labels, #labels = tuple(bar_names)
           ncol=ncol, mode=None,
           borderaxespad=0.,
           loc='center',        # the location of the legend handles
           handleheight=None,   # the height of the legend handles
           #fontsize=9,         # prop beats fontsize
           markerscale=None,
           frameon=False,
           prop=fontprop
           #fancybox=True,
           )

    return fig,leg_fig,leg
```

TODO: add example in tests

`autobasedoc.autoplot.`**`autoPdfImg`** (*func*)

decorator for the autoplot module

returns one PdfImage objects if wrapped plt-function obeys the principle demonstated in following minimal example:

```
@autoPdfImg
def my_plot(canvaswidth=5): #[inch]
    fig, ax = ap.plt.subplots(figsize=(canvaswidth,canvaswidth))
    fig.suptitle("My Plot",fontproperties=fontprop)
    x=[1,2,3,4,5,6,7,8]
    y=[1,6,8,3,9,3,4,2]
    ax.plot(x,y,label="legendlabel")
    nrow,ncol=1,1
    handles, labels = ax.get_legend_handles_labels()
```

(continues on next page)

(continued from previous page)

```
    leg_fig = ap.plt.figure(figsize=(canvaswidth, 0.2*nrow))

    ax.legend(handles, labels, #labels = tuple(bar_names)
           ncol=ncol, mode=None,
           borderaxespad=0.,
           loc='center',         # the location of the legend handles
           handleheight=None,    # the height of the legend handles
           #fontsize=9,          # prop beats fontsize
           markerscale=None,
           frameon=False,
           prop=fontprop
           #fancybox=True,
           )

    return fig
```

`autobasedoc.autoplot.`**`full_extent`**(*ax*, *pad=0.0*)

Get the full extent of an axes, including axes labels, tick labels, and titles.

3.5.2 autorpt

Class AutoDocTemplate customized for automatic document creation this inherits and partly redefines reportlab code

class `autobasedoc.autorpt.`**`AutoDocTemplate`**(*filename*, *onFirstPage=(<function _doNothing>, 0)*, *onLaterPages=(<function _doNothing>, 0)*, *onLaterSPages=(<function _doNothing>, 0)*, *templates=[]*, *leftMargin=14.173228346456693*, *rightMargin=14.173228346456693*, *topMargin=42.519685039370074*, *bottomMargin=42.519685039370074*, *title=None*, *author=None*, *subject=None*, *producer=None*, *creator=None*, *keywords=[]*, *pagesize=(595.2755905511812, 841.8897637795277)*, *debug=False*)

Bases: `reportlab.platypus.doctemplate.BaseDocTemplate`

This is our Document Template, here we want to add our special formatting, that we need for our Document Creation

We derive from BaseDocTemplate, we don't want to and cannot superclass here…

We have one unique way inside afterFlowable to create one outline entry for each flowable by:

- adding the <a name=outlineName> to text within the paragraph flowable.
- using self.notify(), self.canv.bookmarkPage() and self.canv.addOutlineEntry() inside a function afterFlowable() which is overloaded in an own Template class that inherits from BaseDocTemplate

For a basic example, please see in TOC with clickable links by rptlab. we implemented a special action flowable class called Bookmark, that has no actions! please see modifications in afterFlowable(). Unlike with paragraphs we are also abled to store many outline entries from one Table. This is the best way to do this for multible bookmarks in one table flowable without having to write too much overhead code.

We discovered this workaround that was postet in ReportLab-users Group back in 2004 by Marc Stober. This workaround suggests using a class Bookmark. We have ported this to use an ActionFlowable, to do the book-

marking, because this flowable is not doing anything at all, but storing some values, so that afterFlowable() can see them.

To define Template settings for your pages you can use three stages, override the default values:

```
onFirstPage=(_doNothing, 0)
onLaterPages=(_doNothing, 0)
onLaterSPages=(_doNothing, 0)
```

The stages are:

- *onFirstPage* template for the first page
- *onLaterPages* template of any following page
- *onLaterSPages* template on any later switchable special page

Instead of *_doNothing*, a function that just carries a pass, you can define your own function that can draw something on the canvas before anything else gets drawn. For the most common cases there are the following 'template' functions:

- `drawFirstPortrait()`
- `drawFirstLandscape()`
- `drawLaterPortrait()`
- `drawLaterLandscape()`
- `drawLaterSpecialPortrait()`
- `drawLaterSpecialLandscape()`

To define a scheme, where all pages are Portrait but special pages are landscape pages:

```
onFirstPage=onFirstPage
onLaterPages=onLaterPages
onLaterSPages=onLaterSPages
```

addPageInfo(*typ='header'*, *pos='l'*, *text=None*, *image=None*, *line=False*, *frame='First'*, *addPageNumber=False*, *rightMargin=None*, *leftMargin=None*, *topMargin=None*, *bottomMargin=None*, *shift=None*)

add header and footer elements, that raster into the frame:

```
    l            c            r

+---------------------------------+
| l  header                     r |
| e +------------+------------+ i |
| f |                         | g |
| t |                         | h |
|   |                         | t |
| p |                         |   |
| a |                         | p |
| d |                         | a |
|   |                         | d |
|   +------------+------------+   |
|    footer                       |
+---------------------------------+
```

Parameters

- **frame** – "First"/"Later"
- **typ** – header/footer
- **pos** – "l"/"c"/"r"
- **text** – ""
- **image** – preferrably a PDF Image
- **addPageNumber** – True/False
- **rightMargin** – shift of 'r' ancor towards centre (only for line).
- **shift** – shifts the image in (x,y) direction (only for image).
- **line** – True/False if true, draw a line from 'l' to 'r'

creates a PageInfo object: PageInfo(typ,pos,text,image,line,frame,addPageNumber,rightMargin,shift)

As of now we consider either text or image and optionaly a line and will not handle these cases separately.

afterFlowable(*flowable*)

Registers TOC entries. and outline entries

Entries to the table of contents can be done either manually by calling the addEntry method on the TableOfContents object or automatically by sending a 'TOCEntry' notification in the afterFlowable method of the DocTemplate you are using. The data to be passed to notify is a list of three or four items containing a level number, the entry text, the page number and an optional destination key which the entry should point to. This list will usually be created in a document template's method like afterFlowable(), making notification calls using the notify() method with appropriate data.

createFrame(*frame_id='Portrait'*, *x1=0.0*, *y1=0.0*, *width=0.0*, *height=0.0*, *left_padding=0.0*, *bottom_padding=0.0*, *right_padding=0.0*, *top_padding=0.0*, *overlap=None*)

Frame reportlab internal signature:

```
        width                     x2,y2
+---------------------------------+
| l  top_padding                r | h
| e +-------------------------+ i | e
| f |                         | g | i
| t |                         | h | g
|   |                         | t | h
| p |                         |   | t
| a |                         | p |
| d |                         | a |
|   |                         | d |
|   +-------------------------+   |
|    bottom padding               |
+---------------------------------+
(x1,y1) <-- lower left corner
```

figcounter()

a simple figure counter, this is a very dirty way to control the number of figures

getFrame(*temp_name=None*, *orientation=None*, *last=False*)

returns frame

to get the master frame inside the pageTemplates

use orientation=None keyarg (default)

frame._x1,frame._y1 frame._width,frame._height frame._leftPadding,frame._bottomPadding frame._rightPadding,frame._topPadding and pagesize: (x,y)

getMultiColumnTemplate(*frameCount=0*, *template_id='LaterL'*, *onPage_template_func=<function _doNothing>*, *pagesize_landscape=False*)

create a TwoColumn Frame

This is customized for landscape format pages. if you want portrait, set pagesizeL to False

Frame vals:

```
x1,
y1,
width,
height,
leftPadding=6,
bottomPadding=6,
rightPadding=6,
topPadding=6,
id=None,
showBoundary=0,
overlapAttachedSpace=None,
_debug=None
```

Template Style for Two Frames:

```
        width          (x2,y2)
+-----------------------------+-----------------------------+
| l  top padding            r | h                           |
| e +---------------------+ i | e +---------------------+   |
| f |                     | g | i |                     |   |
| t |                     | h | g |                     |   |
|   |                     | t | h |     Second          |   |
| p |                     |   | t |     frame           |   |
| a |                     | p |   |                     |   |
| d |                     | a |   |                     |   |
|   |                     | d |   |                     |   |
|   +---------------------+   |   +---------------------+   |
|    bottom padding           |                             |
+-----------------------------+-----------------------------+
(x1,y1) <-- lower left corner
```

getSpecialTemplate(*temp_name='Later'*)

get the next page action flowable template that starts with temp_name

getTemplate(*temp_id=None*, *last=False*, *as_name=False*)

Return first page template with an id that starts with frame_name

handle_flowable(*flowables*)

overriding base method!!!

try to handle one flowable from the front of list flowables.

added a dirty workaround to scale images if their boundingBox exceeds the borders of the frame.

handle_pageBegin()

override base method to add a change of page template after the firstpage. by default: use Later template

scaleImage(*thisImage*, *scaleFactor=None*)

Function to allow user scaling of factor. A scaling greater than 0, lesser than 1 is allowed. By default a

scaling of 0.7071 is applied to thisImage

template_id

templates_maker()

Here we populate the page templates (register page templates and functions) Page templates are separated into three stages:

- on first page
- on later page
- on later special page

using a list, that simplified looks like this:

```
[PageTemplate(id='First',
              frames=[firstFrame0, firstFrame1, firstFrame2],
              onPage=onFirstPage,
              pagesize=self.pagesize),
 PageTemplate(id='Later',
              frames=[laterFrame0, laterFrame1, laterFrame2],
              onPage=onLaterPages,
              pagesize=self.pagesize),
 PageTemplate(id='Special',
              frames=[specialFrame0, specialFrame1, specialFrame2],
              onPage=onLaterSPages,
              pagesize=self.pagesize)])
```

the list is sent to:

```
BaseDocTemplate.addPageTemplates(list)
```

the templates are later accessible by:

```
self.pageTemplates[index]
```

in self.handle_pageBegin() the next page automatically becomes the first 'Later' flowable

to controll switching on the next page template use self.PageNext e.g:

```
nextTemplate = self.doc.getSpecialTemplate(temp_name="")
ar.PageNext(self.contents, nextTemplate=nextTemplate)
```

updatePageInfo(*pI*)

addPageInfo, using the PageInfo type object

Parameters **pI** – PageInfo() object

class autobasedoc.autorpt.**Bookmark**(*title*, *level=0*)

Bases: reportlab.platypus.doctemplate.NullActionFlowable

Utility class to display PDF bookmark.

Parameters

- **title** – Title of the bookmark
- **level** – Level entry in the outline

classmethod cln()

createBookmarkKey()

creates a Bookmark Key using title, level and the identity of this ActionFlowable

id

level

class autobasedoc.autorpt.**BottomSpacer**(*width*, *height*, *isGlue=False*)
Bases: reportlab.platypus.flowables.Spacer

a spacer that fills the current doc unto the bottom

wrap(*availWidth*, *availHeight*)
This will be called by the enclosing frame before objects are asked their size, drawn or whatever. It returns the size actually used.

class autobasedoc.autorpt.**Footer**
Bases: reportlab.platypus.doctemplate.NullActionFlowable

classmethod cln()

id

class autobasedoc.autorpt.**Header**
Bases: reportlab.platypus.doctemplate.NullActionFlowable

classmethod cln()

id

autobasedoc.autorpt.**PageNext**(*contents*, *nextTemplate*)
switch to nextTemplate on next page

autobasedoc.autorpt.**addHeading**(*title*, *sty*, *page*)
demux function that adds bookmark and heading to page list

autobasedoc.autorpt.**doHeading**(*title*, *sty*, *outlineText=None*, *bookmarkFullpage=False*)
function that makes a Flowable for a heading :param title: title of the paragraph :param sty: style for the paragraph :param outlineText: with specifying an "ancortext" we can control the output to the bookmark name in the outline of the PDF

autobasedoc.autorpt.**doImage**(*Img*, *doc*, *titlename*, *sty*)
Here we simplify the process of inserting an Image to the pdf story

Parameters **doc** – an instance of autobasedoc

#TODO make it more general, the doc.width is not reliable

autobasedoc.autorpt.**doTabelOfContents**()
returns toc with 3 customized headings level styles

autobasedoc.autorpt.**drawFirstLandscape**(*canv*, *doc*)
This is the Template of any later drawn Landscape Oriented Page

the frame object is only used as a reference to be able to draw to the canvas

After creation a Frame is not usually manipulated directly by the applications program – it is used internally by the platypus modules.

Here is a diagramatic abstraction for the definitional part of a Frame:

```
        width                    x2,y2
+---------------------------------+
| l  top padding                r | h
| e +-------------------------+ i | e
| f |                         | g | i
| t |                         | h | g
```

(continues on next page)

(continued from previous page)

```
|   |                          | t | h
| p |                          |   | t
| a |                          | p |
| d |                          | a |
|   |                          | d |
|   +--------------------------+   |
|    bottom padding                |
+----------------------------------+
(x1,y1) <-- lower left corner
```

NOTE!! Frames are stateful objects. No single frame should be used in two documents at the same time (especially in the presence of multithreading.

`autobasedoc.autorpt.`**`drawFirstPortrait`**(*canv*, *doc*)
This is the Title Page Template (Portrait Oriented)

`autobasedoc.autorpt.`**`drawLaterLandscape`**(*canv*, *doc*)
This is the Template of any later drawn Landscape Oriented Page

`autobasedoc.autorpt.`**`drawLaterPortrait`**(*canv*, *doc*)
This is the Template of any following Portrait Oriented Page

`autobasedoc.autorpt.`**`drawLaterSpecialLandscape`**(*canv*, *doc*)
This is the Template of any later drawn Landscape Oriented Page

`autobasedoc.autorpt.`**`drawLaterSpecialPortrait`**(*canv*, *doc*)
This is the Template of any following Portrait Oriented Page

`autobasedoc.autorpt.`**`getBaseFont`**(*fonttype*)

`autobasedoc.autorpt.`**`getBookmarkLast`**(*contents*)
return last bookmark in contents or None

`autobasedoc.autorpt.`**`reprFrame`**(*frame*)

3.5.3 pageinfo

class `autobasedoc.pageinfo.`**`PageInfo`**(*typ*, *pos*, *text*, *image*, *line*, *frame*, *addPageNumber*, *rightMargin=None*, *leftMargin=None*, *topMargin=None*, *bottomMargin=None*, *shift=None*)

Bases: `object`

a Class to handle header and footer elements, that raster into a frame:

```
l    c    r
```

Parameters

- **`typ`** – header/footer
- **`pos`** – 'l'/'c'/'r'
- **`text`** – text of header or footer element
- **`image`** – image preferrably as pdf Image
- **`addPageNumber`** – True/False

As of now we consider either text or image, and will not handle these cases separately.

`autobasedoc.pageinfo.addPlugin(canv, doc, frame=None, talkative=False)`

holds all functions to handle placing all elements on the canvas...

Parameters

- **canv** – canvas object
- **doc** – AutoDocTemplate instance
- **frame** – template name of the Frame

This function suggests that you have stored page info Items in doc.pageInfos.

By default if there is no frame='Later' option, we use the frame='First'

If you don't want to decorate your Later pages, please define an empty pageTemplate with frame='Later'

3.5.4 fonts

`autobasedoc.fonts.registerFont(faceName, afm, pfb)`

Helvetica BUT AS AFM

The below section is NOT equal to:

```
_baseFontName  ='Helvetica'
_baseFontNameB ='Helvetica-Bold'
_baseFontNameI ='Helvetica-Oblique'
_baseFontNameBI='Helvetica-BoldOblique'
```

we will mapp afm files from matplotlib with pfb files from reportlab

this will give embedded Type1 Face Fonts

`autobasedoc.fonts.setFonts(typ)`

Sets fonts for standard font-types

Parameters **typ** (*str*) – one of sans-serif-afm, serif (sans-serif is default on init)

`autobasedoc.fonts.setTtfFonts(familyName, font_dir, normal=(None, None), bold=(None, None), italic=(None, None), bold_italic=(None, None))`

Sets fonts for True Type Fonts

3.5.5 pdfimage

Pdf and Svg Images can be embedded from file, matplotlib drawings can be embedded as file-like objects and behave like a Flowable.

SVG images can currently only be returned as a Drawing:

```
from reportlab.graphics.shapes import Drawing

reportlab.lib.utils.ImageReader
```

class `autobasedoc.pdfimage.PdfAsset(fname, width=None, height=None, kind='direct')`

Bases: `reportlab.platypus.flowables.Flowable`

read in the first page of a PDF file from file

it can e used like a *reportlab.platypus.Flowable()*

drawOn (*canv*, *x*, *y*, *_sW=0*)
Tell it to draw itself on the canvas. Do not override

wrap (*width*, *height*)
This will be called by the enclosing frame before objects are asked their size, drawn or whatever. It returns the size actually used.

class autobasedoc.pdfimage.**PdfImage** (*filename_or_object*, *width=None*, *height=None*, *kind='direct'*)

Bases: reportlab.platypus.flowables.Flowable

PdfImage wraps the first page from a PDF file as a Flowable which can be included into a ReportLab Platypus document. Based on the vectorpdf extension in rst2pdf (http://code.google.com/p/rst2pdf/)

This can be used from the place where you want to return your matplotlib image as a Flowable:

```
img = BytesIO()

fig, ax = plt.subplots(figsize=(canvaswidth,canvaswidth))

ax.plot([1,2,3],[6,5,4],antialiased=True,linewidth=2,color='red',label='a curve')

fig.savefig(img,format='PDF')

return(PdfImage(img))
```

drawOn (*canv*, *x*, *y*, *_sW=0*)
translates Bounding Box and scales the given canvas

wrap (*availableWidth*, *availableHeight*)
returns draw- width and height

convenience function to adapt your image to the available Space that is available

autobasedoc.pdfimage.**convert_px_to_pdf_image_obj** (*img_path*)
convert png image to pdf byte object output can be passed to PdfImage

autobasedoc.pdfimage.**form_xo_reader** (*imgdata*)

autobasedoc.pdfimage.**getScaledSvg** (*path*, *factor*)
get a scaled svg image from file

autobasedoc.pdfimage.**getSvg** (*path*)
return *reportlab.graphics.shapes.Drawing()* object with the contents of the SVG specified by path

autobasedoc.pdfimage.**scaleDrawing** (*drawing*, *factor*, *showBoundary=False*)
scale a reportlab.graphics.shapes.Drawing() object, leaving its aspect ratio unchanged

3.5.6 styledtable

class autobasedoc.styledtable.**StyledTable** (*gridded=False*, *leftTablePadding=0*, *hTableAlignment=None*, *colWidths=None*)

Bases: object

data object to store all data and styles of ONE table

This class is an independent representation for the metadata and parameters of the results to be shown in ONE table:

```
+----------------------------------+
|            table                 |
|                                  |
+----------------------------------+
```

addDoubleLine(*color='blue'*, *line=0*)
Adds a double line below the line specified

Parameters

- **line** – the number of the line below which the double line is inserted
- **color** (*str*) – the color of the vertical line

addHorizontalLines(*color='blue'*, *offsetCol=None*, *exclude=[]*)
Adds horizontal lines only. Apply after inserting all data and after inserting the header line. If no header line exists also a top line is included.

Parameters **color** (*str*) – the color of the horizontal line

addTableExtraStyleCommand(*cmd*)

addTableHeader(*line*, *fonttype='bold'*, *color='blue'*)
Prepends a table line to data and insert the correct styles, like double underline and bold.

Parameters

- **line** (*tuple, list*) – the data for the header line
- **fonttype** (*str*) – one of normal, bold, italic
- **color** (*str*) – the color of the horizontal line

addTableLine(*line*)
Adds a table line to data. If leftTablePadding is activated an empty column is inserted here automatically.

Parameters **line** (*tuple, list*) – the data of one table line

addTableStyleCommand(*cmd*, *extra=False*)
Adds style command to table.

line commands are like: op, start, stop, weight, colour, cap, dashes, join, linecount, linespacing

op is one of: GRID, BOX,OUTLINE, INNERGRID, LINEBELOW, LINEABOVE, LINEBEFORE, LINEAFTER

cell commands are like ...

Parameters **cmd** (*tuple*) – one table command data for styles

addVerticalLine(*col*, *color='blue'*)
Adds vertical lines only. Apply after inserting all data.

Parameters

- **col** (*int*) – the number of the column after which the vertical line is inserted
- **color** (*str*) – the color of the horizontal line

as_flowable
Shortage name

colsCount()
Returns the number of columns.

columnWidthEstim(*data*)
Returns minimum column width for all lines in the column.

Parameters **data** (*list of list*) – the table data

Returns list of cell width estimations

getTableHeight(*frameInfo*)
Returns height of table hint

Parameters **table** (*Table*) – the table object

handleStyleCommands()
Creates real tableStyle from tableStyleCommands.

layoutFullWidthTable(*frameInfo, hTableAlignment=1, marginSide=51.023622047244096, ratios=[0.3, 0.4, 0.3]*)
Returns a table flowable that spans to frame width.

Parameters **styledTable** (StyledTable) – a styled table

Returns a table flowable element

layoutStyledTable(*hTableAlignment=None, colWidths=None, spaceBefore=None, spaceAfter=None, rightPadding=0, pre=False, factor=1*)
Returns a table flowable with automatically estimated column width.

Parameters

- **styledTable** (StyledTable) – a styled table
- **hTableAlignment** (*int*) – the table alignment on the frame
- **colWidths** (*list*) – the columns width in cm, flexible width is -1
- **spaceBefore** (*float*) – the space above the table in cm
- **spaceAfter** (*float*) – the space below the table in cm

Returns a table flowable element

layoutTable(*hTableAlignment=None, colWidths=None*)
Returns a table flowable with automatically estimated column width.

Parameters

- **styledTable** (StyledTable) – a styled table
- **hTableAlignment** (*int*) – the table alignment on the frame
- **colWidths** (*list*) – the columns width in cm
- **spaceBefore** (*float*) – the space above the table in cm
- **spaceAfter** (*float*) – the space below the table in cm

Returns a table flowable element

linesCount()
Returns the number of rows/lines including header line.

setFontSizeColor(*size, color, row, col*)
FONTSIZE (or SIZE) - takes fontsize in points; leading may get out of sync. TEXTCOLOR

setTableData(*data*)
Overwrites the table data with fresh new data. If leftTablePadding is activated an empty column is inserted here automatically.

Parameters **data** (*list of tuples*) – the new data

shift_background_styles(*n*)

sign(*x*)

snip_background_styles(*n*)

split_table(*n*)

split_table_iterative(*frameInfo*, *availableHeight*)

widthEstim(*obj*, *name='table'*)

Estimates the width of an object. If obj is of type str: estimates the width of the text on page, using the self.font If object is of type reportlab.Flowable, then return the result of minWidth()

Parameters

- **obj** (*object*) – the object which will be estimated
- **name** (*str*) – the name of the fontszize

Returns the estimated width

autobasedoc.styledtable.**getTableStyle**(*tSty=None*, *tSpaceAfter=0*, *tSpaceBefore=0*)

Parameters

- **tSty** – TableStyle(tSty) default is None
- **tSpaceBefore** – space before table, default: 0
- **tSpaceAfter** – space after table, default: 0

Returns TableStyle object

use the add method of that object to add style commands e.g.: to add a background in the first row:

```
tableStyle.add(("BACKGROUND",(0,0),(2,0),ar.color_dict().get("green")))
tableStyle.add(("BACKGROUND",(2,0),(4,0),ar.color_dict().get("lavender")))
```

to change text color on the first two columns:

```
tableStyle.add(("TEXTCOLOR",(0,0),(1,-1),ar.color_dict().get("red")))
```

to change alignment of all cells to 'right':

```
tableStyle.add(("ALIGN",(0,0),(-1,-1),"RIGHT"))
```

to add a grid for the whole table:

```
tableStyle.add(("GRID",(0,0),(-1,-1),0.5,ar.color_dict().get("black")))
```

you can use the fixed colors from the reportlab module, but it is better for further customization to use color_dict().get("color_name") instead.

some further examples of command entries:

```
("ALIGN",(0,0),(1,-1),"LEFT"),
("ALIGN",(1,0),(2,-1),"RIGHT"),
("ALIGN",(-2,0),(-1,-1),"RIGHT"),
("GRID",(1,1),(-2,-2),1,ar.colors.green),
("BOX",(0,0),(1,-1),2,ar.colors.red),
("LINEABOVE",(1,2),(-2,2),1,ar.colors.blue),
```

(continues on next page)

(continued from previous page)

```
("LINEBEFORE",(2,1),(2,-2),1,ar.colors.pink),
("BACKGROUND", (0, 0), (0, 1), ar.colors.pink),
("BACKGROUND", (1, 1), (1, 2), ar.colors.lavender),
("BACKGROUND", (2, 2), (2, 3), ar.colors.orange),
("BOX",(0,0),(-1,-1),2,ar.colors.black),
("GRID",(0,0),(-1,-1),0.5,ar.colors.black),
("VALIGN",(3,0),(3,0),"BOTTOM"),
("BACKGROUND",(3,0),(3,0),ar.colors.limegreen),
("BACKGROUND",(3,1),(3,1),ar.colors.khaki),
("ALIGN",(3,1),(3,1),"CENTER"),
("BACKGROUND",(3,2),(3,2),ar.colors.beige),
("ALIGN",(3,2),(3,2),"LEFT"),
("GRID", (0,0), (-1,-1), 0.25, ar.colors.black),
("ALIGN", (1,1), (-1,-1), "RIGHT")
("FONTSIZE", (1,0), (1,0), self.fontsizes["table"])

('SPAN',(1,0),(1,-1))
```

3.5.7 styles

class autobasedoc.styles.**StyleSheet**

Bases: object

This may or may not be used. The idea is to:

1. slightly simplify construction of stylesheets;
2. enforce rules to validate styles when added (e.g. we may choose to disallow having both 'heading1' and 'Heading1' - actual rules are open to discussion);
3. allow aliases and alternate style lookup mechanisms
4. Have a place to hang style-manipulation methods (save, load, maybe support a GUI editor)

Access is via getitem, so they can be compatible with plain old dictionaries.

add(*style*, *alias=None*)

get(*key*, *default=<object object>*)

has_key(*key*)

list()

class autobasedoc.styles.**Styles**

Bases: object

default styles definition

provides a function to easily register more styles

StyleInfoByAlias(*alias*)

prints all attributes of a style matching the alias

addStyle(*PS*, *alias=None*)

add a ParagraphStyle to stylesheet

listAttrs(*style*, *indent=''*)

print all registered styles

listStyles()
return list of styles in object

registerStyles()
register all stylesheets by their respective aliases

3.5.8 tableofcontents

class autobasedoc.tableofcontents.**AutoTableOfContents**
Bases: reportlab.platypus.tableofcontents.TableOfContents

wrap(*availWidth*, *availHeight*)
All table properties should be known by now.

class autobasedoc.tableofcontents.**TocTable**(*data*, ***kwargs*)
Bases: reportlab.platypus.tables.Table

split(*availWidth*, *availHeight*)
This will be called by more sophisticated frames when wrap fails. Stupid flowables should return []. Clever flowables should split themselves and return a list of flowables. If they decide that nothing useful can be fitted in the available space (e.g. if you have a table and not enough space for the first row), also return []

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

a

f

p

s

t

Index

F

G

H

I

L

P

R

S

T

U

W