The PyOpenOffice User Guide - Version 0.4

1. Introduction

1.1. History

I frankly admit that I am not a professional programmer myself: The PyOpenOffice library grew from a *practical* need when I was developing my software "Bezirksreiter". "Bezirksreiter" is a web-based open source software for church administration (member and donation tracking, accounting, reporting and statistics). I was looking for an appropriate reporting tool for my daily needs: serial letters, lists, reports and so on. As I am working as a pastor I often (sometimes too often...) have to do with administrative tasks. I am using OpenOffice.org in my church office (both Linux and Windows version), so I tried to combine "Bezirksreiter" with OO. The basic ideas ("Modifying Existing SXW files") developed, and you still find "Bezirksreiter" code in some parts of PyOpenOffice.

During the time a "spin-off"-product was created from this: The PyOpenOffice library. Other features were added, some of it from my own ideas, other as a respond to user feedback. You still can see the church background in the examples of this user guide: I think PyOpenOffice should not be ashamed of its roots.

A big step was done when I discovered the "tiny" rml2pdf converter from Fabien Pinckaers. A more XML oriented approach was born, you find this in the "New Way" to create PDF and HTML files.

Nevertheless it is still to be developed. ***Which of you Python and XML gurus out there would help to develop PyOpenOffice? It is hosted with SourceForge and can be developed in a team via CVS. I also appreciate all kind of feedback: bug reports, suggestions, code snippets.***

Yes - I should not forget to give you a hint to find "Bezirksreiter" using PyOpenOffice. There is a "live" reference implementation on my homepage: *http://www.bezirksreiter.de/bezirksreiter/verwaltungMain.htm*

Sorry - it's only available in German (who will help to port it to other languages?). Login as "Benutzername" = "Onkel_Otto" and "Passwort" = "12345".

1.2. Thanks

This tool is dedicated to the developers of OpenOffice.org, Python, the Reportlab Toolkit, the Python Imaging Library and "tiny" rml2pdf / rml2html. They did a great job and I admire them for their programming skills.

1.3. Requirements

- Any operating system that supports the following software:
- Python 2.3 (maybe it works form 2.1 upwards, but this is not recommended) for the SXW-Processing, see: *www.python.org*
- The ReportlabToolkit and the Python Imaging Library for the transformation from SXW to PDF , see: *www.reportlab.org* and *www.pythonware.com/products/pil/*
- OpenOffice.org for viewing and editing the templates, see: *www.openoffice.org*
- The PyOpenOffice-Zip Package, see: *www.bezirksreiter.de*
- Any W3C conformant XSLT processor (for the new XSLT based functions)
- To get the latest (unstable) version browse the CVS tree on SourceForge: *http://cvs.sourceforge.net/viewcvs.py/pyopenoffice/pyopenoffice*

1.4. Installation

- Unzip PyOpenOffice.zip into your path
- Make sure that all the skripts have write access to this path
- Write your skript and import pyopenoffice. Mind: there are still global variables in use, so you always have to use the whole module

1.5. Contact and Feedback

The quality of PyOpenOffice depends on your feedback. Bug reports, feature requests, suggestions etc. please mail to: *me.simon@web.de* Homepage: *www.bezirksreiter.de*

1.6. License

PyOpenOffice is now distributed under the Lesser GNU Public License (LGPL), see: *www.gnu.org/copyleft/lesser.html*

1.7. Basic Use

All OO files are a zip archive, consisting of a number of XML files and additional data (embedded bitmap pictures for example). If you want to modify a file, you always have to use the same procedure: unpack - modify one ore more members of the archive - repack. *If you want to open one of the unpacked xml files in an editor you should save your sxw file with "pretty printing". See menu "Extra - Options - Load/Save - General" in OO.*
The OO file format is an open file format and is documented on hundreds of pages (!) in detail. Enjoy it on: *http://xml.openoffice.org/general.html*
The PyOpenOffice tool is tested only for the OO Writer format ("sxw"), nevertheless some of the functions (for example: modifying existing sxw files) should also work with OO Calc etc.
The following examples all assume the PyOpenOffice skripts are in your path. It is planned to make PyOpenOffice more accessable like an UNIX command line tool, but at the moment almost every function of PyOpenOffice can only be accessed by the Python API. The instantiation of "tool" also will create a empty, "dummy" DTD named "office.dtd" in your path - this is needed for the XML parsers.

```
import pyopenoffice
tool = pyopenoffice.PyOpenOffice()
```

2. Modifying Existing SXW Files

2.1. Introduction

Let's think you are running a Christian bookshop and have some data from a database query. To insert the data into your sxw file you have to provide it as a list of dictionaries:

```
lod = [{"Book": "Holy Bible, \nLuther Version", "Price": "15.00"},
       {"Book": "Calendar for Daily Devotion", "Price": "8.00"},
       {"Book": "Holy Bible, Good News Version", "Price": "17.00"},
       {"Surname": "Simon", "FirstName": "Martin"},
       {"Surname": "Miller","FirstName": "Steve"},
       {"Surname": "Smith", "FirstName": "John"},
       {"StrangeField": "StrangeValue"}]
```

Now it is time to discover the makeSerialLetters function. The complete definition of this function is:

```
def makeSerialLetters(self,sourcefile,replace_listofdicts,
one_file=1,replace_fixeddict=textutilities_pyopenoffice.defaultkeywords,
content="",data_with_linebreaks=0,transform_linebreaks=0,RML=0,
with_multi=1)
```

You see three required arguments and a number of switches with default values:

- with one_file=1 the output is merged to one file (see "Serial Letters")
- replace_fixeddict=textutilities_pyopenoffice.defaultkeywords: provides constants like "TFA-today-TFE" (which will insert the current date) and can be modified by the user.
- with "content" = any_UTF-8_string you can provide a preprocessed content.xml you extracted before from the sxw sourcefile.
- with data_with_linebreaks=1 your carriage returns in the replace_listofdicts can be represented in the generated sxw file: This may be useful when processing databases containing values with carriage returns. All CR signs are recognized (\n,\r and \r\n) and replaced by the open office tag <text:line-break/>
- with transform_linebreaks=1, the open office tag <text:line-break/> is replaced by generating a new paragraph (necessary for PDF transformation of this tag).
- with RML = 1 you can process files in the Reportlab Markup Language (RML, see below)
- with_multi = 1 you scan for the Multi:TFA- ... - TFE tag before you make serial letters from the TFA- ... - TFE tag

2.2. Reports and Lists: The Multi:TFA- ... -TFE Tag

This is a function to multiply table rows containing this tag. *These Multi:TFA- ... -TFE tags are always processed before the "normal" TFA- .. -TFE tags (see "Serial Letters")!*
Imagine you have the following table in your sxw template file named "test.sxw":

Book this background colour is set with paragraph background	**Price** this background colour is set with table cell background.
Multi:TFA-Book-TFE	*Multi:TFA-Price-TFE €*

Now you do the following operation:

```
file =  tool.makeSerialLetters(sourcefile="test.sxw", replace_listofdicts=lod,
one_file=1, data_with_linebreaks=1, transform_linebreaks=0, RML=0,with_multi=1)
```

A new file named "1Copies_test.sxw" will be generated, the variable "file" containing the name of this file. Opening it with OO you will find the table above being replaced by the following:

Book this background colour is set with paragraph background	**Price** this background colour is set with table cell background.
Holy Bible, *Luther Version*	*15.00 €*
Calendar for Daily Devotion	*8.00 €*
Holy Bible, Good News Version	*17.00 €*

Mind: obviously only one row with "Multi..." tags should be used in the template table!
Notice how the carriage return before "Luther Version" is replaced by an OO linebreak. Also notice how the keys from the "lod" like "Surname" (that do not fit to any tag names) are automatically dropped by makeSerialLetters.

2.3. Serial Letters: The TFA- ... - TFE Tag

The price list is generated, now we want to send it to some of our customers as a serial letter. For this purpose we use the TFA- ... - TFE tag in the letterhead as follows:

TFA-FirstName-TFE TFA-Surname-TFE

We perform the same operation as above:

```
file =  tool.makeSerialLetters(sourcefile="test.sxw", replace_listofdicts=lod,
one_file=1, data_with_linebreaks=1, transform_linebreaks=0, RML=0,with_multi=1)
```

Now a new file named "3Copies_test.sxw" will be generated, the variable "file" containing the name of this file. There are three identical letters in this three pages file, the TFA- ... - TFE tags are replaced with the dictionaries containing "Surname" and "FirstName" as keys (this is what will happen when you try this with the "test.sxw" that comes with the PyOpenOffice distribution):

..... Martin Simon ..... (remaining part of the first letter) - *Pagebreak*
..... Steve Miller ..... (remaining part of the second letter) - *Pagebreak*
..... John Smith ..... (remaining part of the third letter)

Sometimes you want to have your *serial letters in separate files*. Use the switch "one_file = 0" to do this. The variable "file" now will be a list "['1_test.sxw', '2_test.sxw', '3_test.sxw']", containing the names of all generated files.
The makeSerialLetters function is "clever" enough to always process Multi:TFA-... tags first, and then make serial letters from the "normal" TFA-... tags. Also mind: the dictionary {"StrangeField": "StrangeValue"} - which does not match in our template - is automatically dropped by makeSerialLetters.
The makeSerialLetters function normally works very fast and robust. It does not use any XML parser, but is working with Regular Expressions and search and replace operations. I was reported it is able to process even sxw files in Japanese, some trials with the new OASIS file format of OO 2.0 also succeeded. There is only *one restriction: it will not work with frames that are "anchored at the page".* The workaround is simple: use the "anchor at paragraph" option in your template file instead.

2.4. Catalogues etc.: The Picture Replace Function

Sometimes you have a number of picture files on your server (for example products you are offering in your webstore), and you want to present them in an appealing form. As an example we now want to generate a worksheet for a sunday school lesson. The first step would be to create a template file "test_pic.sxw" with dummy pictures and replacement fields (see your PyOpenOffice distribution):

TFA-Headline-TFE

TFA-Pic1-TFE This picture will be larger than the others. PyOpenOffice takes the width of the dummy picture and makes the width of the inserted picture to be the same. The height of the inserted picture is now calculated from its original proportions.	TFA-Pic2-TFE	TFA-Pic3-TFE

TFA-Pic4-TFE	TFA-Pic5-TFE	What will happen when you provide no replace file? TFA-Pic6-TFE

Assume you have the following lists: "piclist" (containing a list of picture files in the order you want to use them - you have got the files with your PyOpenOffice copy) and "pictest" (a list of dictionaries containing the titles for the pictures and the headline):

```
piclist = ["pray1.pcx","read.pcx","think.pcx","pray2.pcx","act.pcx"]
pictest = [{"Headline": "Useful Bible Study", "Pic1": "Pray", "Pic2": "Read",
"Pic3": "Think", "Pic4": "Pray again", "Pic5": "Practice it!", "Pic6": "You see:
The place is left empty."}]
```

We first use the function replacePictures:

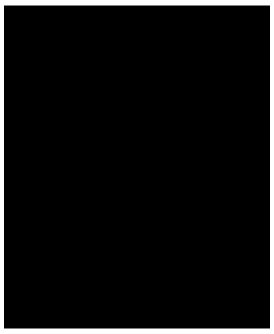
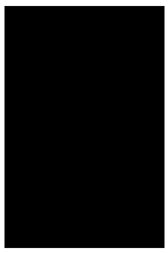
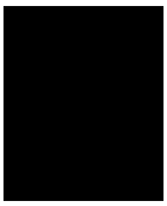
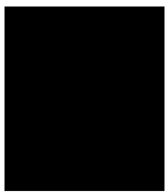
```
file1 = tool.replacePictures("test_pic.sxw",piclist)
```

"file1" will contain the name of the new generated file "NewPics_test_pic.sxw". Then we apply the function makeSerialLetters to the new generated file:

```
file2 = tool.makeSerialLetters(file1,pictest)
```

We get a one copy "serial letter" as a result: "1Copies_NewPics_test_pic.sxw". The result will look like this:

Useful Bible Study

		
Pray This picture will be larger than the others. PyOpenOffice takes the width of the dummy picture and makes the width of the inserted picture to be the same. The height of the inserted picture is now calculated from its original proportions.	Read	Think
		
Pray again	Practice it!	What will happen when you provide no replace file? You see: The place is left empty.

There is one restriction: The pictureReplace function uses the Python Imaging Library. This library does not process all image formats (see the documentation of the PIL or better: just try it out).

3. Creating PDF (The "Old Way")

3.1. Deprecated, but still useful...

From the beginning PyOpenOffice was designed to create PDF's from sxw files without using OO. This is useful for example on a webserver where you cannot easily install and use OO to produce PDF's on demand. Nevertheless the built-in function of OO to export PDF will always be superior for high quality PDF's that represent the exact layout in a WYSIWYG manner. Use the OO export when you need this *and do not consider PyOpenOffice as a WYSIWYG tool for PDF exports - you always will have to live with certain restrictions.*
When I say "the old way" creating PDF's I mean the way this transformation was done until version 0.31: A hardcoded way, using the SAX parser and the XML DOM API from the Python Standard Library, transforming the sxw file to Reportlab "Platypus" objects, which then are used to produce PDF. This way is proved to be stable and it may still be the fastest way to do it.
Nevertheless the code is very hard to maintain and it is not easy to add new features or to adapt it to new file formats. So I decided version 0.31 will be the last version using this way, new versions with new features will use a different and much more flexible approach (see below: "Going XML"). You still can use this proven way in this and the coming versions of PyOpenOffice, but *you will have to live with the following restrictions (some of them are still found in the "new" way version 0.4)*:

3.2. Restrictions

- only the "automatic-styles" of the current document are parsed - inherited styles from master documents etc. will be dropped
- all fonts are mapped to the three standard postscript fonts of PDF: Times, Helvetica and Courier. Only "pt" is accepted for fontsizes
- no page-numbering function
- headers of long tables will not be repeated on every page
- the "underline"-function of the ReportlabToolkit has several bugs. When difficulties occur, PyOpenOffice will change all underlines to *italics.* If text alignment "justified" is used in the document, this will be done by default. *I strongly recommend not to use underlines at all. Underlines are considered as "old fashioned" and ugly in professional typography (have a look at the way how LaTeX does!), reminding us to the good old times of the typewriter... Underlines are not supported in the "new" way.*
- the "underline"-function should not be used with coloured background (else you will see a little white point after the underlined word)
- do not use frames and columns - the text within is printed as normal paragraphs (and sometimes it is just dropped...)
- If at least one cell has borders, the whole table is printed with a grid
- not all image formats are processed (due to the use of the Python Imaging Library)
- image sizes may only be given either in cm or inch
- only headings form heading 1 to heading 3 can be mapped - they will have always a certain standard appearence

3.3. Usage

When you want to deliver a serial letter from an sxw template in PDF format, you will have to perform two steps:

```
file =  tool.makeSerialLetters(sourcefile="test.sxw", replace_listofdicts=lod,
one_file=1, data_with_linebreaks=1, transform_linebreaks=1, RML=0,with_multi=1)
```

"file" will contain the string value "3Copies_test.sxw" (see above). *Mind the switch "transform_linebreaks = 1"*: All OO linebreaks are transformed to separate paragraphs in the output sxw file. This is necessary because the Reportlab Toolkit does not provide intra-paragraph linebreaks. If you need separate files do it as shown above ("Modify Existing SXW Files") with the switch "one_file = 0". Now we call the function makeSafePdf where *"file" can be one filename or a list of filenames*:

```
file2 = tool.makeSafePdf(file)
```

"file2" will contain a list of the generated PDF files, in this example it is ['3Copies_test.pdf']. makeSafePdf performs several trials, when an error occurs: full featured, without underlines, without images, text only without all formatting as a last rescue. So it is a really safe function - you will always get a readable output from an SXW file.

4. Going XML - The Future of PyOpenOffice

4.1. Introduction

As I already said the possibilities of the "Old Way" are robust but very restricted:

- You can transform your SXW files only to one format (PDF). What will you do if you want your server to deliver them in HTML?
- It is very difficult to transform more than a very simplified layout from SXW to PDF. What will you do when you need more style information?
- What will you do when you want to transform the coming OO OASIS file format to PDF? I am not very keen on rewriting a great part of my code...
- ..... ?

So what I am doing now is the following:

Step 1a		
	PyOpenOffice	
sxw	➜	"normalized" sxw file
Step 1b		
	XSLT	
"normalized" sxw file	➜	RML
Step 2		
	insert your data with PyOpenOffice (Serial Letters, Reports and Lists, Picture Replace)	
RML	➜	NEW_RML
Step 3		
	trml2pdf	
NEW_RML	➜	PDF
	trml2html	
NEW_RML	➜	HTML

Remarks:

1. RML is the "Report Markup Language" (see below), a special kind of XML to describe pages. It is similar to XSL-FO, but much more usable.
2. trml2pdf and trml2html are python scripts written by Fabien Pinckaers. Have a look at his homepage: *http://www.openreport.org*

As you can see you some additional programs are required: an XSLT processor, a suitable XSLT stylesheet, and two python scripts named "trml2pdf" and "trml2html". This sounds complicated, but in practice this is much clearer and easier to maintain than a hardcoded only approach. **PyOpenOffice 0.4 contains an sxw preprocessor, an XSLT stylesheet to generate RML from "normalized" sxw, and an RML-to-PDF processor. Future versions also may contain an RML-to-HTML processor.**

4.2. "Normalizing" the SXW format

After unpacking an sxw archive you will find two files named "content.xml" and "styles.xml". *The style information for formatting the text is contained in both files and is build using a complex style hierarchy.*

Example for a paragraph style named "P11":

```
       derived from                derived from
P11 <-------------- TextBody <-------------- Standard
```

In some way it is working similar to class hierarchies in object oriented programming languages. Many styles you find in an OO document are derived from a so called "parent style". To find a font size of a certain style, OO calculates this way:

```
120%                          (nothing)                            10pt
P11   <----------------- TextBody <------------------- Standard
found in                        found in                           found in
content.xml                     styles.xml                         styles.xml
```

Got it? => This will result in a font size of 12pt for paragraph style "P11". To make your XSLT transformation easier, PyOpenOffice does some preprocessing:

```
n = tool.unpackNormalize("test.sxw")
```

A new file named "normalized_test.xml" is stored, additional information from "test.sxw" like included picture files is unpacked into the same path. The information in "normalized_test.xml" is merged from information in content.xml and styles.xml, and will contain only absolute values for every style. All units like "cm" or "inch" are converted to points (72pt = 1 inch). => In "normalized_test.xml" the style "P11" from our example will now have a font size of "12.0" (without the "pt" after it).
As XSLT is not a language that is very suitable for complex calculations this preprocessing will dramatically decrease the complexity of your XSLT stylesheets.

4.3. What is RML?

RML is an XML language to describe text which is formatted mainly for printing. Nevertheless you can transform it not only to PDF, but also to HTML. Maybe in future other transformations to RTF, LATEX etc. also will be possible. The purpose of RML is similar to XSL-FO, but in my opinion it is much clearer and more powerful. RML was developed and defined by Reportlab to make it easier to work with the Reportlab Toolkit. Instead of generating a lot of Python objects to describe your PDF document (the "Old Way" of PyOpenOffice), you write an RML document and give it to an RML-to-PDF processor which will generate the corresponding Python objects for you. Have a look at: *http://www.reportlab.com/eprs_index.html*
Reportlab combined this with a commercial, enterprise level solution for PDF reports. You will find in it very ambitioned features like PDF encryption or a "page catcher" to integrate existing PDF documents into your RML file. When you do not need such high level features and are looking for an open source solution, you will appreciate the "tiny" rml2pdf processor written by Fabien Pinckaers. It implements almost all RML functions needed for text and graphics formatting and will fit for the needs of PyOpenOffice. To learn RML when using the "tiny" rml2pdf processor you can use the original "RML User Guide" written by Reportlab (it is mirrored on the PyOpenOffice homepage) - just skip the sections for the enhanced features of the commercial solution.
"Oh no, not that! Will I now have to learn a complex XML language to use PyOpenOffice?" Don't be afraid - PyOpenOffice can hide the complexity of RML from you if you want. Design your documents with OO and let PyOpenOffice do the job for you. Nevertheless - if you want to do some fine tuning or you need features of RML that are not implemented yet by PyOpenOffice, you can deal with the RML source code to improve it.

4.4. Using XSLT to transform "normalized" SXW files

We now have to transform "normalized_test.xml" to the RML language. A good choice to do XML-to-XML transformations is the platform independent XSLT language defined by the W3C. An XSLT processor transforms an XML file using a so called "stylesheet". PyOpenOffice comes with a stylesheet "normalized_oo2rml.xsl". This is the first version of it and will need a lot of improvement, of course - nevertheless it is considered to be stable, and it is already able to process more style information than the hardcoded "Old Way" of PyOpenOffice (hey, you XML gurus out there - work on it and let me know your ideas!).
You will need an W3C conformant XSLT processor installed on your system. I do not want to force you to use a special one - this will depend on your personal preferences, your environment where you are using PyOpenOffice, your operating system etc. I made myself good experiences with the libxml / libxslt library which was developed for the GNOME project. All major LINUX distributions should contain a copy of it, a Python API for it is also avaible. It is written in C, very fast and very conformant to the W3C standard. For my example I will use it not with the Python API, but use it as a LINUX command line tool started by the Python interpreter:

```
import os
os.system ("xsltproc normalized_oo2rml.xsl normalized_test.xml > test.rml")
```

Adapt this example to your XSLT processor and your operating system. *Mind: When starting the XSLT processor from the commandline your generated file "test.rml" will not be tracked by the PyOpenOffice cleanup routine (see below).*

4.5. Modifiying Existing RML Files

One of the most important features of PyOpenOffice is modifying existing SXW files to generate reports, lists, serial letters, catalogues with pictures etc. Now imagine you have serial letter with 200 recipients. When you first make a serial letter from it in the sxw format and then do the "unpackNormalize" / XSLT transformation cycle on this new, large sxw file, you will find it working *very* slow. The reason is that those transformations are quite "expensive" compared to the makeSerialLetters function - "unpackNormalize" and XSLT both are complex XML operations using recursions etc.
Also you lack the possibility to edit your RML templates manually when you need special features of RML. Example: You want to insert a letterhead using the "pageGraphics" option of RML (this cannot be done automatically by PyOpenOffice yet), and you want to appear this letterhead on all your generated serial letters.
For this reason PyOpenOffice is able to do all operations "Modifying Existing SXW Files" also on RML files. Use them for files automatically generated by PyOpenOffice, or use them for hand-written RML files (for example when you like RML, but you do not like OO). Use the TFA- ... -TFE and Multi:TFA-...-TFE tags the way described above, and do your picture replace operations the same way. Insert the needed tags into the sxw file, transform it to RML - or insert the tags manually into the RML sourcecode.
When inserting pictures manually into the RML sourcecode mind that PyOpenOffice only scans for images that are enclosed by an <illustration> tag. Have a look at the "make_image" template in the XSLT stylesheet to understand it in more detail. When you are using automatically generated RML files for picture replacements, you do not have to think about it - PyOpenOffice will do everything for you appropriately.

Modifying RML files is straightforward:

```
file =  tool.makeSerialLetters(sourcefile="test.rml", replace_listofdicts=lod,
one_file=1, data_with_linebreaks=1, transform_linebreaks=1, RML=1,with_multi=1)
```

Just set the switch "RML=1". You can drop the switch "transform_linebreaks" - it is always set to "1" automatically when you use the RML option. Do picture replacement the same way:

```
piclist = ["pray1.pcx","read.pcx","think.pcx","pray2.pcx","act.pcx"]
file1 = tool.replacePictures("test_pic.rml",piclist,RML=1)
```

4.6. Example: Manual Editing of an RML File Created by PyOpenOffice

We now take the file "test.rml" and want to improve it a bit. I show you how to insert a "pageGraphics" section - you find it in the file "pageGraphics_test.rml" in your PyOpenOffice distribution:

```
<?xml version="1.0"?>
<document filename="test.pdf">
  <template pageSize="(595.0,842.0)" title="Test" author="Martin Simon"
allowSplitting="20">
    <pageTemplate id="first">
      <frame id="first" x1="42.0" y1="30.0" width="511" height="782"/>
      <pageGraphics>
        <setFont name="Helvetica" size="200.0" />
        <fill color="lightgrey" />
        <translate dx="200" dy="200" />
        <rotate degrees="45.0" />
        <drawString x="100" y="100">Test</drawString>
      </pageGraphics>
    </pageTemplate>
  </template>
  <stylesheet>
    <blockTableStyle id="Standard_Outline">
    .............
```

The code inserted manually is highlighted. This will print very large grey letters "Test", rotated by 45 degrees to the left, as a background on all the pages of your document. When you make serial letters from this template, every letter will contain this background. Examples for more special features of RML you find on Fabiens homepage: *www.openreport.org*

4.7. Using RML to create PDF

When you like the commercial tools from Reportlab, you now can use them to transform one of the RML files mentioned above - see the description in the "RML User Guide". Here I describe the "poor man's open source solution" as an alternative. For this purpose I use a patched version of "tiny" rml2pdf. There where very few patches necessary to adapt it to the needs of PyOpenOffice: Now it is possible to use German "Umlaute" and the € - sign in your documents. Also all your tables are now attached to the left margin of your page (and not centered on the page anymore). If you do not like these patches it is easy to remove them - have a look at the sourcecode of "patched_trml2pdf.py". *Be aware that you also have the additional files in your path that are needed by patched_trml2pdf.py: colors.py, utils.py, __init__.py.*
You can use the PDF converter from the command line (see the sourcecode). We use it from PyOpenOffice here, using the same function like in the "Old Way". *Mind the switch "RML=1".*

```
list_of_outfiles = tool.makeSafePdf(oowriter=file_or_listoffiles, RML=1)
```

When an error occurs, list_of_outfiles will be an empty list. Maybe this will change in future.

4.8. Using RML to create HTML

This is a feature not included yet in the PyOpenOffice package. Please, have a look at the homepage of trml2html when you need this: *www.openreport.org*

4.9. Putting it all together

Has PyOpenOffice become too difficult to use now? I do not think so. This will be the example for the normal cycle producing a serial letter when you are using the "New Way":

```
import pyopenoffice
import os
tool = pyopenoffice.PyOpenOffice()
n = tool.unpackNormalize("test.sxw")
# now n =  "normalized_test.xml"
os.system ("xsltproc normalized_oo2rml.xsl normalized_test.xml > test.rml")
file =  tool.makeSerialLetters(sourcefile="test.rml", replace_listofdicts=lod,
one_file=1, data_with_linebreaks=1, transform_linebreaks=1, RML=1,with_multi=1)
list_of_outfiles = tool.makeSafePdf(oowriter=file_or_listoffiles, RML=1)
```

You see: just two more steps than the "Old Way" - but with all advantages of the XML approach. Allright?

5. Miscelleanous Other Functions

Here I only speak only about features you can access directly with the PyOpenOffice object. Have a look at the sourcecode if you want to have access to more functions.

5.1. Creating Plain Text Files and PDF's without any Formatting

Sometimes you want a plain text representation of an sxw file. PyOpenOffice can do this for you: All XML-tags for formatting are dropped, no images are processed, table cells are transformed to paragraphs with text data. You can choose plain text output (*set plain_text_output=1*) or PDF output:

```
list_of_outfiles = tool.makePlainText(file_or_listoffiles,plain_text_output=0)
```

5.2. Extract the content.xml file from the sxw archive

With this function you can extract the content.xml from an SXW-Archive. Use this to work on it with your own customized functions, then pass the preprocessed content.xml to the variable "content" of makeSerialLetters(). *Always keep in mind that everything in content.xml has to be UTF-8-encoded. Do not use the RML switch: is designed for internal purposes only.*

```
content_string = tool.extractContentXml(self,sourcefile,RML=0)
```

5.3. Dealing with the files generated by PyOpenOffice

Working with PyOpenOffice you will soon find a lot of files in your path. PyOpenOffice keeps track of them in the (not human readable) file "PyOpenOffice_Tempfiles". List them all with:

```
list = tool.listTempfiles()
```

Then make your great spring-cleaning and remove them all:

```
tool.cleanTempfiles()
```

Now you are done. Relax and switch off your machine...