

Quality PDF Documents: Part II

by John Redmond

formatted December 12, 2009

CONTENTS

Including Non-Default \LaTeX Packages	1
Adding Non-Standard Packages to Your \LaTeX Distribution.	1
Styling the Headers	1
Another Table Option	2
Another Image Option	3
Building eBook Documents	4
Character Encoding	5
A Rabbit from the Hat	6
Under the Wraps	6
Working with Pure XML Source	7
Stylesheets are Simple	9

This document builds on Part I by introducing standard ways of enhancing a \LaTeX distribution with specialised packages.

INCLUDING NON-DEFAULT \LaTeX PACKAGES

Many styling packages are supplied in the standard TeX Live distributions. If we need to use any of them, we must specify this in a *package* statement:

```
package { select: color, multicol; }
package { select: fancyhdr, caption, textcomp, float, alltt;}
```

ADDING NON-STANDARD PACKAGES TO YOUR \LaTeX DISTRIBUTION.

Very many other packages are available through the \LaTeX community. I have attempted to delay including them in the earlier discussion, but some of them are seriously useful (as we shall soon see). Additional packages are supplied as text (ASCII) files. These are pasted into your \LaTeX tree in the usual way. On my Linux system, \LaTeX is installed at the following path:

```
/usr/share/texmf/tex/latex
```

I created a new directory (called 'extra', but it could be anything sensible) inside the *latex* directory and copied a few extra package files into the new directory. A little fiddly, but not something that has to be done often. To use techniques in this extended tutorial, it is necessary to add these packages inside the *latex* directory (such as in the *extra* directory) and then to execute the following command:

In Linux (as superuser):

```
texhash
```

In Windows (when the \LaTeX installation in the C:\directory):

```
\texlive\bin\win32\texhash
```

After you have done this, your \LaTeX package knows about the additional packages and you can use them with your L exx ia installation.

STYLING THE HEADERS

As you might expect, \LaTeX has powerful and very precise ways of specifying the display of headers: space above, space below, indentation, font details, etc (also in terms of lengths), but it is not something that will attract a beginner. But, as ever, help is at hand with an additional package, *titlesec*. It can be used at two levels: *basic* and *extended*. L exx ia incorporates the basic interface, with a quite dramatic effect on the appearance of a document—particularly an *article* document, such as one of the Shakespeare plays. The stylesheet for the plays includes:

ANOTHER TABLE OPTION

```
package { titlesec; }
headings { font-size: large; font-shape: upright; font-weight: bold; text-align:
center; display: extended; }
```

The *headings* specifications are important here, and their settings should be pretty obvious. Change one of them, and the appearance of the whole document is suddenly different. The options are:

```
font-size: large, Large, LARGE;
font-shape: upright, slanted, italic, small-caps;
font-weight: normal, bold;
text-align: center, left, right;
display: compact, extended;
```

The headings in the present document were set using *titlesec*. They are not exactly to my taste, but they demonstrate some of the options. They were set by:

```
headings { font-size: large; font-shape: small-caps; font-weight: bold; text-align:
center; display: extended; }
```

It is a good idea to experiment with these variants to see how they fit with your document—and there may be some surprises. The main one might be the response to font-size, in that silly choices are likely to be ignored.

Note how easy it is to specify centered headings with *titlesec*. Without it, centering is somewhat tricky; in fact, default headings are resolutely left-aligned (which happens to work very well with *book* documents, but not so well for *article* documents).

ANOTHER TABLE OPTION

My preferred option for tables is the *tabulary* package, because it seems to cope better with tables that have very wide columns. There is very little new to learn about it, because it is so similar to *tabularx*. (And that is not surprising because both packages were written by the same author.) The only difference is the way in which the columns are specified (it could not be simpler):

L, C and R denote left, center and right alignment.

There is no need to indicate potentially problem (wide) columns and everything seems to flow into place. As it happens, it is now possible to have wide columns center-aligned. This is fine for any table headings or footers but, for extended text, the results are probably not what you had in mind.

To use it, you need the *tabulary* package in your L^AT_EX distribution and the following declarations in your external CSS stylesheet:

```
document { table-style: tabulary; }
lengths { tymin: 0.15\tablewidth; tymin: 0.4\tablewidth; }
```

ANOTHER IMAGE OPTION

Table 1: Table with Specified Alignments

Heading 1	Heading 2	heading 3	Heading 4
Content	More content	Longer content	The longest content of all in this column
Row 2 Content	More content	Longer content	The long content in this column
Content 3	More content	Longer content	A somewhat longer content in this column
Content 4	More content	Longer content	The longest content of all the rows in this column
Foot 4	More Foot	Foot 3	Foot 4

Using *tym*in and *tymax* gives some control over the minimum and maximum widths of columns, and is worth experimenting with. (Note that they are *lengths*, and that they are conveniently defined as proportions of the *tablewidth*, which is already defined in the external stylesheet. Note too that you cannot set these widths *without* specifically setting the table-style to *tabulary*, because these widths are defined inside the *tabulary* package.)

To revisit our silly table (from Part I), we change the column alignment settings:

```
_table
@caption=Table with Specified Alignments
@label=DEMO1
@cols=R|CC|C
_thead=Heading 1|Heading 2|heading 3|Heading 4
Content|More content|Longer content|The longest content of all in this column
Row 2 Content|More content|Longer content|The long content in this column
Content 3|More content|Longer content|A somewhat longer content in this column
Content 4|More content|Longer content|The longest content of all the rows in this
column
_tfoot=Foot 4|More Foot|Foot 3|Foot 4
```

So *tabulary* is there as an option if you feel that you need it.

ANOTHER IMAGE OPTION



We have seen that an image can be included inline or, better, as a centered float with a title/caption and label. With the extra *picins* package, it is possible to specify *@align* as *left* or *right* and have the following text flow around it (as here). Note that the image is still pretty much inline with the text content. To get this effect, it is necessary to include in the external CSS stylesheet:

```
package { picins; }
```

This option can be very elegant and professional; and it permits adding a caption, but ignores a label. Moreover, it is your responsibility to use it thoughtfully. It is not permissible to flow text between *both* a left- and a right-aligned image. And it is a mistake to include such an image directly before a section heading or structure like a table or list. More subtly, side images work best when there are sufficiently long blocks of text to wrap around them (as in the present paragraph). And it is a mistake to position a left-aligned image directly before a section heading. Some of the worst results of flowing small blocks of text around images are illustrated in an accompanying PDF file ([./aesop.pdf](#)). This file also demonstrates a general requirement that the images be roughly the same in size—and that centered images should have a landscape format. Finally, as with tables, images can occur at inconvenient points near page breaks, so that a satisfactory final document may need several attempts to resize and relocate images.



FIGURE 1:
QANTAS

BUILDING EBOOK DOCUMENTS

The developing interest in eBooks and their document formats provides a real opportunity for production on PDF files of specific page dimensions. \LaTeX and L^aT_EX are well positioned to adapt to whatever page sizes and shapes become standard, with a styling specification of the following type:

```
geometry { param: paperwidth=125mm, paperheight=175mm, hmargin=15mm,
vmargin=20mm; }
```

together with an enlarged font (here, 12 point on a 13-point line separation):

```
fonts { palatino: 12/13, ptm, m, n; }
```

One of \LaTeX 's resoundingly unmatched strengths is its display of fully justified and automatically hyphenated text. Nothing else comes close but a, perhaps unexpected, problem emerges with such a combination of large font size and narrow line width: \LaTeX occasionally cannot find a good basis for a line break, with the result that there is a rather ugly overshoot at the right margin. But a solution is at hand:

- The first solution is definitely *not* a solution: turn off automatic hyphenation with *text-align=left* in the document specification. It is very enlightening to actually try this, as it demonstrates how ugly a ragged right margin can be;

CHARACTER ENCODING

- The real solution is to allow \LaTeX to be more relaxed about how it achieves justified right margins. Again, in the document specification, set *fussy*: *false* (it is *true* by default). This allows \LaTeX to generate wider gaps between words and still retain a good-looking full justification.

The full specification of an eBook in Lexxia follows. In view of the limited display real estate, the page headers for the body are limited to a single center heading (the chapter title) and the chapter headings are reduced in size and centered. There is a note in the text source, which is not displayed in the PDF document. Importantly, the paper width and height can be altered to accommodate any display device.

```
document { fussy: false; }
geometry { param: paperwidth=125mm, paperheight=175mm, hmargin=15mm,
vmargin=20mm; }
headers { toc: nyn; body: nyn; }
headings { font-size: small; font-shape: small-caps; font-weight: bold; text-align:
center; display: compact; }
fonts { palatino: 12/13, ppl, m, n; }
versestyle { define: style; environment: quotation; font: helvetica; font-size:
footnotesize; text-shape: normal; }
headerstyle { font-size: small; font-weight: bold; font-shape: slanted; }
verse { style: versestyle; }
L2 { display: page; }
note { display: none; }
```

CHARACTER ENCODING

If you are reworking existing text, there is some chance that non-English characters will turn up. For example, there is a smattering of accented characters in the `sherlock.txt` file that I provide. To what extent you can identify and show these characters depends on your editor. Again, the Linux editors perform pretty well: it is possible to display the characters, but entering them at the keyboard is another matter. The problem is that \LaTeX traditionally requires input files that are kosher ASCII, and it has its own, pretty effective, ways of representing accented European characters.

Unicode and UTF encodings now abound and it is now fair to say that the use of 16-bit characters has not proved the best option. When creating or editing new text source should always use UTF-8 encoding—if only because it is backwards compatible with ASCII. So, when saving your source file and where you have a choice (Linux again!) you should choose UTF-8.

Now, when you come to process such a file with \LaTeX , and it has some of those extended characters, expect to get a complaint. To get things working, you need:

- To have added the *inputenc* package (the file is `inputenc.sty`) to your \LaTeX distribution;
- To add the package to your stylesheet file:

UNDER THE WRAPS

```
document { encoding: utf8; }
```

Actually, it is not a bad idea to have this declaration in any file that you are setting up, because you have a bet each way: you can cope with extended characters and you can still define accented characters in the traditional L^AT_EX way (look to the L^AT_EX wiki for how you would do this).

To really understand the point of all this, you should start to think about some of those technical and perhaps confusing terms, such as ASCII, 8-bit ASCII, latin1, latin2, etc, Unicode, UTF-8, UTF-16, etc. For more information than you will ever need, try <http://en.wikipedia.org/wiki/UTF-8> and http://en.wikipedia.org/wiki/ISO/IEC{}_8859-1. In short, there have been all sorts of short-cuts and bad policy decisions over 20+ years. The best advice now is to try very hard to work only with UTF-8!

If, on the other hand, you are forced to work with material that is in, say, latin-1 encoding, *and your editor cannot save it as UTF-8*, specify in your external CSS stylesheet:

```
document { encoding: latin1; }
```

A RABBIT FROM THE HAT

All of the above is very prescriptive—and likely to be somewhat mysterious. How is this happening? For some insight, try something tangential:

```
lexxia thisfile.txt -w -O*.html
```

where “thisfile.txt” is the text source of the pdf file that you are looking at. You have just built an HTML file; so look at it in a browser. You have not been wasting your time outside the mainstream of XML text processing. Now, we can go just a little further with our tangent, by specifying an CSS/XHTML stylesheet:

```
lexxia thisfile.txt -Wpage.css -O*.html
```

Now we have a browser document with included style specifications. This is all possible because the text documents that we have been talking about are loaded by Lexxia into a DOM documents that are really just a dressed-up XML files with XHTML-type elements.

So, if you are thinking of trying your hand at Lexxia and PDF, look in your bottom drawer for any XHTML files that you might have and try processing them. If they are old-style nasty HTML documents with unterminated tags, you might as well still try, but no promises. Why? Because unterminated tags mean ill-defined structure, which is why HTML is dying.

UNDER THE WRAPS

Details of how to use Lexxia are given in the LimpidSoft site www.limpidsoft.com. I provide here just a simple overview of how Lexxia processes source files into L^AT_EX format:

The “-L” command is a composite of four primitive Lexxia commands:

-1: Analyse a source file and build a Schema document;

WORKING WITH PURE XML SOURCE

- 22: Inspect the Schema document and collate all specified paths, extract subpaths of length 2. build a document listing these subpaths and the special properties of elements with these paths;
- 4xxx.css: Using the internal CSS stylesheet, updated by the external stylesheet (xxx.css), build an XSLT stylesheet. Store this XSLT stylesheet internally. If you are processing multiple source files in a single command, these first three steps are applied only to the first file;
- T: Use the internal XSLT stylesheet to transform the initial source document, and any subsequent source documents (think of Shakespeare's plays), to \LaTeX documents and output these documents.

To understand fully what is happening here, you can execute the individual steps and save them to the screen or files and examine them in detail:

```
lexxia source.txt -1 -O*.xsd // get a Schema file (source.xsd)
lexxia source.txt -1*.dtd // get a DTD file (source.dtd)
lexxia source.txt -1 -22 -O*-inspect.xml // get an inspection file source-inspect.xml
lexxia source.txt -1 -22 -4mystyle.css -O*.xsl // get an XSLT stylesheet file
source.xsl
lexxia source.txt -Tsource.xsl+*.tex // get the source.tex file
```

In occasional cases, it can be useful to generate the inspection file. You may even find that you need to edit it before using it to generate the XSLT file. You would then generate the stylesheet file:

```
lexxia source-inspect.xml -4mystyle.css -Osource.xsl
```

WORKING WITH PURE XML SOURCE

It should now be obvious that a dottedfile text file is converted by Lexxia to an internal XML document. To date, this document has been processed further in light of *specific named tags*. How these tags are handled is determined by their actual *names*, such as *p* or *table*. This is conceptually pretty simple, but Lexxia can work in a more analytical way, by classifying the tags without attaching any significance to their names. This is how it can do a rather nice job of formatting the XML source of a Shakespeare play, such as *hamlet.xml*. It manages to do well because the play is *highly structured*. When we use the following command

```
lexxia source.txt -1 -22 -O*-inspect.xml
```

the result is an XML analysis document:

```
<ELEMENTS>
<ELEMENT path="/PLAY" type="L1"/>
<ELEMENT path="/PLAY/TITLE" type="T1"/>
```

```

<ELEMENT path="/PLAY/PERSONAE" type="L2"/>
<ELEMENT path="PERSONAE/TITLE" type="T2"/>
<ELEMENT path="PERSONAE/PERSONA" type="D2"/>
<ELEMENT path="PERSONAE/PGROUP" type="L3"/>
<ELEMENT path="PGROUP/PERSONA" type="D3"/>
<ELEMENT path="PGROUP/GRPDESCR" type="E3"/>
<ELEMENT path="/PLAY/ACT" type="L2"/>
<ELEMENT path="ACT/TITLE" type="T2"/>
<ELEMENT path="ACT/SCENE" type="L3"/>
<ELEMENT path="SCENE/TITLE" type="T3"/>
<ELEMENT path="SCENE/STAGEDIR" type="D3"/>
<ELEMENT path="SCENE/SPEECH" type="L4"/>
<ELEMENT path="SPEECH/SPEAKER" type="F4"/>
<ELEMENT path="SPEECH/LINE" type="M4"/>
<ELEMENT path="LINE/STAGEDIR" type="m5"/>
<ELEMENT path="SPEECH/STAGEDIR" type="S4"/>
</ELEMENTS>

```

Here we have all the unique two-step paths in the original document, together with their deduced types. To understand these types is to understand how Lexxia works:

- L1, L2, etc:** the level-1, level-2, etc blocks, corresponding to PLAY, PERSONAE, ACT.
- T1, T2, etc:** the descriptive tags of each of these blocks. These are all *singletons* and at the *starts of the blocks*;
- E3:** an end feature tag, which is *always a singleton and always at the end of a block*;
- S1, S2, etc:** other *singletons*;
- F3:** a feature tag, similar to T3, but *not a singleton*. (Some of the SPEECHes have multiple SPEAKERs);
- M4:** a mixed-content tag, consisting of text with *embedded tags*;
- m5:** an embedded tag, as part of mixed content;
- D1, D2, etc:** default tags.

With these classifications in mind, it is possible to do a good job of setting up an XSLT stylesheet, in light of the CSS stylesheets, to transform the original hamlet.xml file to good L^AT_EX. The name of an original tag irrelevant, because the only thing that matters is its *type*, as in the following snippets of the default (internal) CSS stylesheet:

STYLESHEETS ARE SIMPLE

```
T1, author, ignore { display: none; }
T2 { style: section; }
T3 { style: subsection; }
T4, F3 { style: featurestyle; }
m1, m2, m3, m4, m5, m6, m7, m8 { style: emph; display: inline; }
```

In overview, Lexxia allows you to place bets in either way: named tags or structure—or *a mixture of the two!* This is possible because, if a tag is one of the Lexxia styling tags, it is not classified any further, so that the *type* of the path is set to the *name* of the tag. For example, if there had been an element path of *SCENE/note*, it would have been classified as:

```
<ELEMENT path="SCENE/note" type="note"/>
```

Possibly confusing, but certainly powerful and flexible.

At risk of the bleeding obvious, note that the Shakespeare plays process so well because of what they are: a literal, sequential rendering of the contents of the plays into XML format. This means that not all XML files are amenable to direct conversion; they may need prior transformation to a sequential format. Easy if you know XSLT!

STYLESHEETS ARE SIMPLE

The assignment of *tag types* makes styling generic and not reliant on the actual *tag names* in a source document. This means that, for the most part, the external stylesheets deal only with the fine details of presentation. This is illustrated by the following two stylesheets: the first is for Grimms' fairy tales (which works on a text document with *tag names*):

```
document { class: book; toc-columns: 2; body-columns: 1; title: true; pdf: true; }
headers { style: headerstyle; toc: nyn; body: nyn; }
T2 { style: chapter; }
T3 { style: section; }
headerstyle { font-size: small; }
subtitlestyle { define: style; environment: center; font-weight: bold; font-shape:
slanted; }
plainstyle { define: style; display: page; font: courier; font-size: small; text-align:
left; }
note { style: plainstyle; }
comment { style: subtitlestyle; }
verse {style: versestyle; }
```

and the second for the Shakespeare plays (which are *XML documents with structure*):

STYLESHEETS ARE SIMPLE

```
document { class: article; font: times; text-align: left; body-columns: 2; title-page:
true; author: "William Shakespeare"; date: styled; pdf: true; }
package { titlesec; }
headings {font-size: large; font-shape: small-caps; font-weight: bold; text-align:
center; display: compact; }
D3 { style: capsstyle; }
```

A final small, but very useful, point is how to ignore a tag completely, with no display. There is a *note* section at the start of the grimm.txt file. To ignore it, specify:

```
note { display: none; }
```