Generate TEX documents using pdfscript

Oleg Parashchenko

Abstract

Generation of correct TEX files is actually a hard task with a number of peculiarities. Therefore, it is better to delegate this task to some library or tool. A tool already exists (TEXML); now it's time for a library.

The library pdfscript helps to create TEX files from Python. The API follows the LATEX model: it represents environments, commands and their parameters as calls of the corresponding functions in the library.

The pdfscript interface can be used as a basis for object-oriented abstractions of document elements, so that the users may create PDF documents having no idea that TEX is inside.

1 Introduction

Automatic generation of TEX files is much harder than one might expect. Here are some cases where bugs are possible and attention is required:

- Special symbols should be escaped
- Non-latin letters should be handled
- A space after command names may be required: `\it` *text*, not `\it`*text*
- An empty group after a command may be required: `\PDF{} file`, not `\PDF file`
- Opening and closing curly braces should be balanced
- It is necessary to comment-out empty lines to avoid false paragraph breaks (and do you know for sure what an empty line is?)

It is easy to code all these requirements, but at the next level, when we have several different TEX-generating programs, we would like to put the code into a common library. I tried it and found that this is a challenging task, which required a lot of thinking and several attempts before the result was satisfactory.

To compare the result with existing approaches, I asked about related work in the newsgroup comp.text.tex [3]. Surprisingly, the only alternatives are the use of the "print" statements and templates. When a programmer generates TEX files, he surely develops some helper functions, but so far nobody has shared his experiences, or at least I failed to find such work.

In this article, I describe my steps in designing a TEX generation library named pdfscript. Then I use the library to re-typeset an excerpt from "Essential LATEX" by Jon Warbrick [5] and show the artifacts of refactoring the code. Finally, I make a summary of the pdfscript API and speculate on further development.

The "proof of the concept" implementation of the pdfscript library and the examples from this article are available from http://uucode.com/download/pdfscript-article-examples-20100909.tar.gz. Despite its experimental status, the code is ready for use by early adopters.

2 Designing the interface

This section describes the steps of the design process.

2.1 Sample TEX document

Let's start with a very simple document, which contains only a setup, a title and a few paragraphs. (For editorial reasons, the boilerplate text is corrupted by introducing line breaks.)

```
\documentclass[a4paper]{article}
\usepackage[utf8]{inputenc}
\usepackage[T1]{fontenc}
\begin{document}
\section{De finibus bonorum et malorum}
Lorem ipsum dolor sit amet, consetetur sad
ips cing elitr, sed diam nonumy eirmod tem
por invidunt ut labore et dolore...

Duis autem vel eum iriure dolor in hendrer
it in vulputate velit esse molestie conseq
uat, vel illum dolore eu feugiat...

Ut wisi enim ad minim veniam, quis nostrud
exerci tation ullamcorper suscipit loborti
s nisl ut aliquip ex ea commodo...
\end{document}
```

2.2 TEXML version

The first step in the search for an API was to create a TEXML representation. To learn TEXML, visit the homepage of the project — http://getfo.org/texml — or read my *TUGboat* article [4]. For the purposes of this paper, it is enough to know the basics:

- TEXML is an XML format
- The root element is named TeXML
- A TEX command is represented by an element cmd with the attribute name:

```
\command[options]{parameter}
≡
<cmd name="command">
  <opt>options</opt>
  <parm>parameter</parm>
</cmd>
```

- If a command or an environment has options or parameters, they are represented by elements `opt` and `parm`, as in the example above.
- An environment is represented by an element `env` with the attribute `name`:

```
\begin{itemize}
...
\end{itemize}
≡
<env name="itemize">
  ...
</env>
```

This knowledge is enough to rewrite the sample document in the TEXML format:

```
<TeXML>
  <cmd name="documentclass">
    <opt>a4paper</opt><parm>article</parm>
  </cmd>
  <cmd name="usepackage">
    <opt>utf8</opt><parm>inputenc</parm>
  </cmd>
  <cmd name="usepackage">
    <opt>T1</opt><parm>fontenc</parm>
  </cmd>
  <env name="document">
    <cmd name="section">
      <parm>De finibus bonorum et..</parm>
    </cmd>
    <TeXML>Lorem ipsum dolor sit amet, con
      setetur sadipscing elitr, sed diam n
      onumy eirmod tempor invidunt ut l...
    </TeXML>
    <cmd name="par"/>
    <TeXML>Duis autem vel eum iriure dolor
      in hendrerit in vulputate velit esse
      molestie consequat, vel illum dol...
    </TeXML>
    <cmd name="par"/>
    <TeXML>Ut wisi enim ad minim veniam, q
      uis nostrud exerci tation ullamcorpe
      r suscipit lobortis nisl ut aliip...
    </TeXML>
  </env>
</TeXML>
```

The rewriting process is mostly straightforward, but two points require additional comments.

First, the use of the element `TeXML` not only as the root, but also as a container for the text. It is needed here only to satisfy my XML-related experience, which recommends avoiding mixing text and elements without a reason.

Second, in the TEX version, the empty lines give implicit `\par` commands, while TEXML version uses `par` directly. It is possible to generate empty lines, but this is bad style when using TEXML. And by the way, I dislike the version with `par` too. In my documents I prefer to wrap paragraphs to environments and hide `\par` in the environment definitions.

2.3 Direct Python counterpart of the TEXML version

The TEXML version has structured the document, and now it is easy to re-write it in Python:

```
import pdfscript
from pdfscript import opt, parm
doc = pdfscript.newdoc()
doc.cmd('documentclass',
  opt('a4paper'), parm('article'))
doc.cmd('usepackage',
  opt('utf8'), parm('inputenc'))
doc.cmd('usepackage',
  opt('T1'), parm('fontenc'))
indoc = doc.env('document')
indoc.cmd('section',
  parm('De finibus bonorum et malorum'))
indoc.text('Lorem ipsum dolor sit amet, co
  nsetetur sadipscing elitr, sed diam...')
indoc.cmd('par')
indoc.text('Duis autem vel eum iriure dolo
  r in hendrerit in vulputate velit e...')
indoc.cmd('par')
indoc.text('Ut wisi enim ad minim veniam,
  quis nostrud exerci tation ullamcor...')
h = open('30_direct.texml', 'w')
doc.get_root().writexml(h)
h.close()
```

The first line instructs Python to load the library `pdfscript`, the second line allows using the short names `opt` and `parm` instead of the fully qualified `pdfscript.opt` and `pdfscript.parm`.

Then we create a document and put the commands and the environment into it. The content of the article is put inside the environment `document` by attaching the commands to the variable `indoc`, which is associated with the environment.

Finally, we get the root node of the constructed XML document and save it into the file.

2.4 Improved Python code

Immediate exprerience with the code above suggests the following improvements:

- In the most cases, the arguments of `cmd` are the parameters for the command, therefore it is logical to make `parm` calls implicit.
- Instead of `cmd('name', ...)` or `env('name', ...)`, the alias `name('...')` looks better.

- The functions could accept more than one argument.

Implementing these ideas, we get the following Python code:

```
import pdfscript
from pdfscript import opt, par
doc = pdfscript.newdoc()
doc.documentclass(opt('a4paper'), 'article')
doc.usepackage(opt('utf8'), 'inputenc')
doc.usepackage(opt('T1'), 'fontenc')
indoc = doc.document()
indoc.section('De finibus bonorum et ...')
indoc.add('Lorem ipsum dolor sit amet, con
  setetur sadipscing elitr, se...', par())
indoc.add('Duis autem vel eum iriure dolor
  in hendrerit in vulputate ve...', par())
indoc.text('Ut wisi enim ad minim veniam,
  quis nostrud exerci tation u...', par())
h = open('50_final.texml', 'w')
doc.get_root().writexml(h)
h.close()
```

3 Observations on a real world example

To test if the `pdfscript` library is powerful enough, I tried to reproduce some real life LaTeX with it. After wandering in the `doc` directory on CTAN, I decided to re-typeset the document "Essential LaTeX" [5]. Surprisingly, the task was challenging. Even though the LaTeX code contained little markup, it was enough to clutter the Python counterpart. To introduce clarity to the code, a redesign was required.

After some thought, the definition of the notion "clarity" became ambitious: a programmer who has never heard of LaTeX should understand each line of the code. This approach produced a few artifacts:

- Python document classes
- Python macros
- Python active strings

3.1 Python Document Class

Let's consider the high-level structure of the "Lorem ipsum" example:

```
10  doc = pdfscript.newdoc()
20  doc.documentclass(...)
30  doc.usepackage(...)
40  doc.usepackage(...)
50  indoc = doc.document()
60  indoc.section(...)
70  indoc.add(..., par())
80  indoc.add(..., par())
90  indoc.text(...)
```

Let me turn into a non-LaTeX programmer and read this code. Here would be my comments:

(10) Ok, create a new default document. (Wrong!)

(20) This line probably defines the layout and formatting of the document I'm going to create. Why not join (10) and (20)?

(30), (40) Some formatting plugins are loaded. WTF ([2])? What is T1? Do I really need these lines? I do a "Lorem ipsum" example, not something special. If this trivial test requires some functionality, it should be automatically loaded by default.

(50) WTF? I've already created the document, why create it once more?

(60) Good, a section is created. The argument is the title.

(70) Looks like a paragraph is created. But what is this `par()` inside `add()`? Is it an additional vertical space to separate paragraphs, like pressing `<ENTER>` twice in OpenOffice or Word?

(70), (80) Stylistic note. The paragraphs should belong to sections, not to the document itself.

Having these remarks in mind, I recoded the high-level structure in this way:

```
doc = esla.doc()
sect = doc.section(...)
sect.para(...)
sect.para(...)
sect.para(...)
```

The only question about this re-worked code fragment is what does `esla` in the first line mean. A programmer can guess that it is some Python package which assists in creation of the documents and hides the formatting in the commands `section()` and `para()`. Correct. For a LaTeX user, this package is a Python version of a document class or a package. The name `esla` is an abbreviation for "Essential LaTeX"—I'm leaving "Lorem ipsum" test and starting work on the challenging example.

3.2 Python macros

After the high-level structure is improved, time to switch to the inline markup. Here is a fragment of the source code:

```
You then get \LaTeX{} to process the file,
and it creates a new file of typesetting c
ommands; this has the same name as your fi
le but the ``\fn{.TEX}'' ending is replace
d by ``\fn{.DVI}''.  This stands for `{\it
D\/}e{\it v\/}ice {\it I\/}ndependent' ...
```

The direct transcription is a nightmare:

```
sect.para(
 '...You then get ',
 cmd('LaTeX'),
```

```
 ' to process the file, and it creates a
  new file of typesetting commands; this
  has the same name as your file but the',
 verbatim(' ‘‘'),
 cmd('fn', '.TEX'),
 verbatim("''"),
 ' ending is replaced by ',
 verbatim('‘‘'),
 cmd('fn', '.DVI'),
 verbatim("''"),
 '.  This stands for ‘',
 group(cmd('it'), 'D', verbatim('\/')),
 'e',
 group(cmd('it'), 'v', verbatim('\/')),
 'ice ',
 group(cmd('it'), 'I', verbatim('\/')),
 "ndependent' ...")
```

Switching back to code review mode:

- The command `LaTeX` probably produces the logo.
- The lines with `fn` and `it` produce some formatting. But why is the usage different? `fn` has an argument, and `it` is enclosed in a group.
- Well, I think `it` switches to another formatting forever and the group limits this forever. But the construction `\/` makes no sense to me.
- There is too much repetition, I don't like to code that way.

Unifying the `fn` and `it` interface, hiding the details and removing the repetitions, we get a better result:

```
sect.para(
 '...You then get ', cmd('LaTeX'),
 ' to process the file, and it creates a
  new file of typesetting commands; this
  has the same name as your file but the',
 fn('.TEX'), ' ending is replaced by ',
 fn('.DVI'), '.  This stands for ‘',
 it('D'), 'e', it('v'), 'ice ',
 it('I'), "ndependent' ...")
```

The functions `fn` and `it` can be considered as macros, which are expanded in Python, not in LaTeX itself. Incidentally, `pdfscript` implements the `\let\def=\undefined` idea of Jonathan Fine [1].

3.3 Python active strings

There is still an inconvenience. What's easy in LaTeX:

```
... before ... \LaTeX{} ... after ...
```

is being written in Python as:

```
sect.para('... before ... ',
          cmd('LaTeX'),
          ' ... after ... ')
```

For one time use, it is ok. But when we have several LaTeXs in a paragraph, the code looks spaghettiish, syntactically overcomplicated. A simple solution is to let the computer do the low-level job. We can write a function `subst_latex`, which finds the entries of the substring `LaTeX` in the source text and substitutes them with the corresponding commands. With help of this function, the code can be simplified:

```
sect.para(subst_latex(
   '... before ... LaTeX ... after ... '))
```

Nearly all the paragraphs of "Essential LaTeX" contain the logo, therefore it is logical to redefine the function `para`, asking it to call `subst_latex` automatically. The final code is:

```
sect.para(
   '... before ... LaTeX ... after ... ')
```

In this code, the string `LaTeX` can be considered as an active string (by analogue to the active characters), expanded in Python.

At this point, the code does not use `pdfscript` at all. Instead, it communicates only with the `esla` package, which encapsulates not only the formatting details, but also the details of PDF generation.

4 API reference

The previous sections have given enough examples to demonstrate how to use the `pdfscript` library. Here is a more formal description.

For brevity, instead of fully qualified names like `pdfscript.something` I use simple `something`.

4.1 Module functions

`TeXsubdoc newdoc(arg1, arg2, ..., argN)`

Creates a new in-memory document. If there are any arguments (of type `string` or `TeXsubdoc`), they are added to the document. The parts of the documents are constructed with the following functions:

```
TeXsubdoc cmd(name, arg1, arg2, ..., argN)
TeXsubdoc env(name, arg1, arg2, ..., argN)
TeXsubdoc opt      (arg1, arg2, ..., argN)
TeXsubdoc parm     (arg1, arg2, ..., argN)
TeXsubdoc group    (arg1, arg2, ..., argN)
TeXsubdoc text     (arg1, arg2, ..., argN)
TeXsubdoc verbatim (arg1, arg2, ..., argN)
```

These functions create the corresponding elements in TeXML. The library does not validate whether a combination of functions makes sense. Notes on the functions:

- `cmd` and `env` require at least one argument (of type `string`), which is the name.
- String arguments of `cmd` are wrapped with implicit calls of `parm`.

- The functions `text` and `verbatim` create an element `TeXML`. The latter function additionally sets the element's attributes in such a way that the text is passed to TEX without any changes.

4.2 Methods of `TeXsubdoc`

`xml.dom.minidom get_root(self)`

Returns an XML subtree associated with the object `self` of type `TeXMLsubdoc`.

`TeXMLsubdoc add(self, arg1, ..., argN)`

Adds subdocuments `argX` of either type `string` or `TeXMLsubdoc` into the subdocument `self`. Returns the reference to the last added subdocument (`argN`, possibly cast to the type `TeXMLsubdoc`).

```
TeXMLsubdoc cmd  (self, arg1, ..., argN)
TeXMLsubdoc env  (self, arg1, ..., argN)
TeXMLsubdoc opt  (self, arg1, ..., argN)
TeXMLsubdoc parm (self, arg1, ..., argN)
TeXMLsubdoc group(self, arg1, ..., argN)
TeXMLsubdoc text (self, arg1, ..., argN)
TeXMLsubdoc verbatim(self, arg1, ..., argN)
```

The first method is a shortcut for:

```
self.add(cmd(arg1, ..., argN))
```

In this definition, the object method `cmd` uses the module function `cmd` to create a document fragment, and then calls the object method `add` to attach the fragment. The other shortcuts are defined in the same way.

4.3 Aliases

Some commands and environments can be accessed via aliases: `name('...')` instead of `cmd('name', ...)` or `env('name', ...)`. Such aliases are created by the following module functions:

```
register_cmd(name)
register_env(name)
```

5 Conclusion and further work

Despite having no experience yet of `pdfscript` use in a production environment, the experiments so far already allow us to speculate how this tool affects different groups: TEX users, TEX-related developers and the world-outside-TEX.

TEX users can safely ignore `pdfscript`. It is dubious to stop typesetting in TEX and start doing it in Python. As demonstrated by the "Essential LATEX" example, such Python-TEX code is rather unreadable.

I expect that developers writing something-to-LATEX converters will find `pdfscript` useful. The library allows one to concentrate on the main point of the program and not worry about generating correct TEX syntax. Further, representing a future TEX document in a tree simplifies adding refinements, such as changing penalties in the last paragraph of a section.

If TEX could be used as a library, what would its API look like? The `pdfscript` approach is a possibility. First, it is enough. Second, optimizations are possible. Commands could be converted to tokens directly, without serializing first to a string and then parsing this string in TEX. Similar, text content could immediately become TEX characters, without first escaping and then unescaping.

For me, the most important part, however, is how `pdfscript` affects the non-TEX world. In the final version of the "Essential LATEX" example, we saw that sections, paragraphs, inline markup and other document elements are represented as objects with properties and methods. This approach fits perfectly with modern programming practice, and therefore I hope that `pdfscript` will become a viable alternative to XSL-FO and other PDF creation tools. And when one uses `pdfscript`, one is actually using TEX.

The next step of the work is to move from the prototype to a first production version. In particular, I plan to make a PHP version of `pdfscript`, develop a few PHP stylesheets (document templates) and collect users' feedback to decide on the priorities of further development.

References

[1] Jonathan Fine. TEX forever! In *EuroTEX 2005 (Pont-à-Mousson) Proceedings*, pages 140–149, 2006. `http://tug.org/TUGboat/Articles/tb27-0/fine.pdf`.

[2] Alex Papadimoulis. The Daily WTF: Curious Perversions in Information Technology. `http://thedailywtf.com/`.

[3] Oleg Parashchenko. API to generate TEX files, search for related work. `comp.text.tex`, `http://groups.google.com/group/comp.text.tex/browse_thread/thread/ba29ef069a47f00a/`.

[4] Oleg Parashchenko. TEXML: Resurrecting TEX in the XML world. *TUGboat*, 28(1):5–10, March 2007. `http://tug.org/TUGboat/28-1/tb88parashchenko.pdf`.

[5] Jon Warbrick. Essential LATEX. `http://mirror.ctan.org/info/latex-essential/`.

◇ Oleg Parashchenko
bitplant.de GmbH
Fabrikstr. 15
89520 Heidenheim, Germany
`olpa (at) uucode dot com`
`http://uucode.com/`