Getting Started with plasTeX

Tim Arnold

February 19, 2008

1 Overview

This document is part one of a "Getting Started with plasTeX" series to explain how to extend plasTeX so it can understand, parse, and render your customized LaTeX source.

If you have standard 'plain-vanilla' LaTeX files, plasTeX will work for you already (see the main user documentation to use the `plastex` command). And if you have simple customizations, plasTeX can read your package files and work as-is. But when you have more complex styles or classes, you'll need to extend plasTeX to work with your customized files. It's quite easily done: there are two tasks to get started:

1. add a Python class corresponding to each macro you have defined. You'll inherit from a standard plasTeX class; often there's little more to it than that. You define the classes so plasTeX can understand how to parse your new commands.

2. add a template to render the content resulting from plasTex parsing. Your command will have some data or text that needs to be handled in some way in order to display itself correctly, depending on what format you want to render to. The template tells the renderer how to do that.

Once you have those tasks completed, you set your environment PATHs and run plastex. That's all there is to it. We'll do a silly start-up example just to run through the steps. Afterwards we'll do some more interesting things. Note that this example is so simple, plasTeX could easily understand the customization with no action on your part– it can often recognize new commands and deal with their definitions by itself. However, sometimes you'll have new commands or environments that more easily handled by writing your own classes. Let's get started and go through this step-by-step so you can see how plasTeX works.

2 Getting Started: Setting Up

To get started, we'll set up a workspace so we can scale it up with increasingly complex needs and new commands. First, create directories hold the new Python classes and the html templates corresponding to those classes. Also create a corresponding directory to hold containers for the LaTeX packages. That's three new directories:

src/ contains python classes corresponding to new packages
Action: create a file called `mypackage.py`

render/ contains html template files corresponding to new packages
Action: create a file called `mypackage.zpts`

tex/ contains style files corresponding to new packages
Action: create a file called `mypackage.sty`

Finally set three environment variables to tell plasTeX where to look for instructions:

- set PYTHONPATH to include the `src` directory
- set TEXINPUTS to include the `tex` (and current '.') directory
- set XHTMLTEMPLATES to the `render` directory

How you do that depends on what operating system you use. On my machine, the commands look like this:

```
setenv PYTHONPATH      ~/example/src
setenv TEXINPUTS       ~/example/tex:.::
setenv XHTMLTEMPLATES  ~/example/render
```

2.1 Example for Testing: the LaTeX source

To get started, we'll create a new command `myBold` that will render its text as bold. Create a file in the current directory called `testing.tex` with the following lines:

```
\documentclass{article}
\usepackage{mypackage}
\begin{document}
  Hello \myBold{World}
\end{document}
```

2.2 mypackage.sty

In the `tex` directory, write the `mypackage.sty` file to contain these lines:

```
\ProvidesPackage{mypackage}
\newcommand{\myBold}[1]{\textbf{#1}}
```

Note: Recall that at the beginning I said that if you have very simple customizations, plasTeX can read your package files and work as-is. Our example certainly falls into that category. We're only using this as an example to focus on the steps involved in extending plasTeX.

Note: This LaTeX code is not necessary as far as plasTeX is concerned; I include it here since you probably want to use LaTeX to create a pdf or postscript file from your

sources, and LaTeX has to have the definition. But as long as plasTeX can find your definitions (i.e., classes) corresponding to the commands in your source, this file could just as well be blank. In fact, sometimes it's quite useful to spoof plasTeX in that way, but that's a trick for another article.

3 Getting Started: Coding

There are two parts to code: the Python class called `myBold` and the HTML template called `myBold`. These names match the new command we're defining. Since this command takes a single argument, we'll inherit from the plasTeX `Base.Command` class. The only thing the python class needs to do is to parse the command and its arguments so we can get to the data it contains when we need to render it.

3.1 Create the `myBold` class

In mypackage.py:

```
from plasTeX import Base
class myBold(Base.Command):
    args = 'text:str'
    def invoke(self, tex):
        Base.Command.invoke(self, tex)
```

Once plasTeX sees the `myBold` command in your source, it will parse it and the node will contain a string attribute called 'text'. See the main user documentation for details. For now that's really all we need to know–we only need the text argument given to the `myBold` command.

3.2 Create the `myBold` Renderer

The renderer takes the data from the node and renders it according to the template. Here we set the argument in a `b` tag. I like to set the class on my tags to the name of the command they came from.

```
name:myBold
<b class='myBold' tal:content='self/attributes/text'></b>
```

Each node you want to render has the same structure: the `name:` specifies the name of the command you're rendering, followed immediately by a template defining how to render the command and its data.

3.3 Parse and Render with plastex

That's it for setting up and coding. Now plasTeX knows enough to parse the tex file, create a document object and render the nodes of that object.

```
plastex testing.tex
```

4 Results

In your `work` directory, you'll see these entries:

```
testing/   testing.paux   testing.tex
```

The subdirectory `testing` contains a number of things, but for now focus on the main file, `index.html`. That file contains the results of rendering our simple LaTeX source. It was rendered using the `default` theme; inside it you'll see the result of the `myBold` command:

```
Hello <b class="myBold">World</b>
```

We'll talk about the other items in that directory as well as the `testing.paux` file later on.

5 Getting Deeper

To add more functionality in terms of new commands and environments, you add more classes to your package.py file, plus the corresponding render templates in the package.zpts file.

There are a lot of questions remaining; how do you use a different theme, what kinds of things can you do in the template, what is the `zpts` extension mean, how do you get DocBook XML from your LaTeX sources, etc. Future articles will deal with those questions and more.

Comments are welcome: a_jtim at bellsouth dot net