plasTeX — A Python Framework for Processing LaTeX Documents

Kevin D. Smith

7 February 2008

SAS
Email: Kevin.Smith@sas.com

CONTENTS

CHAPTER

ONE

Introduction

plasTEXis a collection of Python frameworks that allow you to process LATEX documents. This processing includes, but is not limited to, conversion of LATEX documents to various document formats. Of course, it is capable of converting to HTML or XML formats such as DocBook and tBook, but it is an open framework that allows you to drive any type of rendering. This means that it could be used to drive a COM object that creates a MS Word Document.

The plasTEX framework allows you to control all of the processes including tokenizing, object creation, and rendering through API calls. You also have access to all of the internals such as counters, the states of "if" commands, locally and globally defined macros, labels and references, etc. In essence, it is a LATEX document processor that gives you the advantages of an XML document in the context of a language as superb as Python.

Here are some of the main features and benefits of plasTEX.

Simple High-Level API The API for processing a LATEX document is simple enough that you can write a LATEX to HTML converter in one line of code (not including the Python `import` lines). Just to prove it, here it is!

```
import sys
from plasTeX.TeX import TeX
from plasTeX.Renderers.XHTML import Renderer
Renderer().render(TeX(file=sys.argv[-1]).parse())
```

Full Configuration File and Command-Line Option Control The configuration object included with plasTEX can be extended to include your own options.

Low-Level Tokenizing Control The tokenizer in plasTEX works very much like the tokenizer in TEX itself. In your macro classes, you can actually control the draining of tokens and even change category codes.

Document Object While most other LATEX converters translate from LATEX source another type of markup, plasTEX actually converts the document into a document object very similar to the DOM used in XML. Of course, there are many Python constructs built on top of this object to make it more Pythonic, so you don't have to deal with the objects using only DOM methods. What's really nice about this is that you can actually manipulate the document object prior to rendering. While this may be an esoteric feature, not many other converters let you get between the parser and the renderer.

Full Rendering Control In plasTEX you get full control over the renderer. There is a Zope Page Template (ZPT) based renderer included for HTML and XML applications, but that is merely an example of what you can do. A renderer is simply a collection of functions[1]. During the rendering process, each node in the document object is passed to the function in the renderer that has the same name as the node. What that function does is up to the renderer. In the case of the ZPT-based renderer, the node is simply applied to the template using the `expand()` method. If you don't like ZPT, there is nothing preventing you from populating a renderer with functions that

[1] "functions" is being used loosely here. Actually, any callable Python object (i.e. function, method, or any object with the `__call__` method implemented) can be used.

invoke other types of templates, or functions that simply generate markup with print statements. You could even drive a COM interface to create a MS Word document.

CHAPTER

TWO

plastex — The Command-Line Interface

While plasTEX makes it possible to parse LATEX directly from Python code, most people will simply use the supplied command-line interface, **plastex**. **plastex** will invoke the parsing processes and apply a specified renderer. By default, **plastex** will convert to HTML, although this can be changed in the **plastex** configuration.

Invoking **plastex** is very simple. To convert a LATEX document to HTML using all of the defaults, simply type the following at shell prompt.

```
plastex mylatex.tex
```

where 'mylatex.tex' is the name of your LATEX file. The LATEX source will be parsed, all packages will be loaded and macros expanded, and converted to HTML. Hopefully, at this point you will have a lovely set of HTML files that accurately reflect the LATEX source document. Unfortunately, converting LATEX to other formats can be tricky, and there are many pitfalls. If you are getting warnings or errors while converting your document, you may want to check the FAQ in the appendix to see if your problem is addressed.

Running **plastex** with the default options may not give you output exactly the way you had envisioned. Luckily, there are many options that allow you to change the rendering behavior. These options are described in the following section.

2.1 Command-Line and Configuration Options

There are many options to **plastex** that allow you to control things input and output file encodings, where files are generated and what the filenames look like, rendering parameters, etc. While **plastex** is the interface where the options are specified, for the most part these options are simply passed to the parser and renderers for their use. It is even possible to create your own options for use in your own Python-based macros and renderers (see in particular Section 5.1.2). The following options are currently available on the **plastex** command. They are categorized for convenience.

2.1.1 General Options

Configuration files

> **Command-Line Options: --config=*config-file*** `or` **-c *config-file***
> **Config File:** [general] config
> specifies a configuration file to load. This should be the first option specified on the command-line. Below is a sample configuration file:

```
[general]
renderer=HTML5
copy-theme-extras=yes

[document]
lang-terms=lang.xml

[files]
split-level=1
```

Kpsewhich
Command-Line Options: --kpsewhich=*program*
Config File: [general] kpsewhich
Default: kpsewhich
specifies the **kpsewhich** program to use to locate LaTeX files and packages.

Renderer
Command-Line Options: --renderer=*renderer-name*
Config File: [general] renderer
Default: XHTML
specifies which renderer to use.

Themes
Command-Line Options: --theme=*theme-name*
Config File: [general] theme
Default: default
specifies which theme to use.

Extra theme files
Command-Line Options: --copy-theme-extras `or` **--ignore-theme-extras**
Config File: [general] copy-theme-extras
Default: yes
indicates whether or not extra files that belong to a theme (if there are any) should be copied to the output directory.

2.1.2 Document Properties

Base URL
Command-Line Options: --base-url=*url*
Config File: [document] base-url
specifies a base URL to prepend to the path of all links.

Number of Columns in the Index
Command-Line Options: --index-columns=*integer*
Config File: [document] index-columns
specifies the number of columns to group the index into.

Language terms
Command-Line Options: --lang-terms=*string*
Config File: [document] lang-terms
Specifies a list of files that contain language terms, delimited by the OS path separator (such as : for POSIX and ; for Windows).

Section number depth
Command-Line Options: --sec-num-depth=*integer*
Config File: [document] sec-num-depth
Default: 6
specifies the section level depth that should appear in section numbers. This value overrides the value of the secnumdepth counter in the document.

Title for the document
Command-Line Options: --title=*string*
Config File: [document] title
specifies a title to use for the document instead of the title given in the LaTeX source document

Table of contents depth
Command-Line Options: --toc-depth=*integer*
Config File: [document] toc-depth
specifies the number of levels to include in each table of contents.

Display sections in the table of contents that do not create files
Command-Line Options: --toc-non-files
Config File: [document] toc-non-files
specifies that sections that do not create files should still appear in the table of contents. By default, only sections that create files will show up in the table of contents.

2.1.3 Counters

It is possible to set the initial value of a counter from the command-line using the **--counter** option or the "counters" section in a configuration file. The configuration file format for setting counters is very simple. The option name in the configuration file corresponds to the counter name, and the value is the value to set the counter to.

```
[counters]
chapter=4
part=2
```

The sample configuration above sets the chapter counter to 4, and the part counter to 2.

The **--counter** can also set counters. It accepts multiple arguments which must be surrounded by square brackets ([ ]). Each counter set in the **--counter** option requires two values: the name of the counter and the value to set the counter to. An example of **--counter** is shown below.

```
plastex --counter [ part 2 chapter 4 ] file.tex
```

Just as in the configuration example, this command-line sets the part counter to 2, and the chapter counter to 4.

Set initial counter values
Command-Line Options: --counter=*[counter-name initial-value]*
specifies the initial counter values.

2.1.4 Document Links

The links section of the configuration is a little different than the others. The options in the links section are not preconfigured, they are all user-specified. The links section includes information to be included in the navigation

object available on all sections in a document. By default, the section's navigation object includes things like the previous and next objects in the document, the child nodes, the sibling nodes, etc. The table below lists all of the navigation objects that are already defined. The names for these items came from the link types defined at `http://fantasai.tripod.com/qref/Appendix/LinkTypes/ltdef.html`. Of course, it is up to the renderer to actually make use of them.

Name	Description
home	the first section in the document
start	same as *home*
begin	same as *home*
first	same as *home*
end	the last section in the document
last	same as *end*
next	the next section in the document
prev	the previous section in the document
previous	same as *prev*
up	the parent section
top	the top section in the document
origin	same as *top*
parent	the parent section
child	a list of the subsections
siblings	a list of the sibling sections
document	the document object
part	the current part object
chapter	the current chapter object
section	the current section object
subsection	the current subsection object
navigator	the top node in the document object
toc	the node containing the table of contents
contents	same as *toc*
breadcrumbs	a list of the parent objects of the current node

Since each of these items references an object that is expected to have a URL and a title, any user-defined fields should contain these as well (although the URL is optional in some items). To create a user-defined field in this object, you need to use two options: one for the title and one for the URL, if one exists. They are specified in the config file as follows:

```
[links]
next-url=http://myhost.com/glossary
next-title=The Next Document
mylink-title=Another Title
```

These option names are split on the dash (-) to create a key, before the dash, and a member, after the dash. A dictionary is inserted into the navigation object with the name of the key, and the members are added to that dictionary. The configuration above would create the following Python dictionary.

```
{
    'next':
        {
            'url':'http://myhost.com/glossary',
            'title':'The Next Document'
        },
    'mylink':
        {
            'title':'Another Title'
        }
}
```

While you can not override a field that is populated by the document, there are times when a field isn't populated. This occurs, for example, in the *prev* field at the beginning of the document, or the *next* field at the end of the document. If you specify a *prev* or *next* field in your configuration, those fields will be used when no *prev* or *next* is available. This allows you to link to external documents at those points.

Set document links

Command-Line Options: --links=*[key optional-url title]*
specifies links to be included in the navigation object. Since at least two values are needed in the links (key and title, with an optional URL), the values are grouped in square brackets on the command-line ([ ]).

2.1.5 Input and Output Files

If you have a renderer that only generates one file, specifying the output filename is simple: use the **--filename** option to specify the name. However, if the renderer you are using generates multiple files, things get more complicated. The **--filename** option is also capable of handling multiple names, as well as giving you a templating way to build filenames.

Below is a list of all of the options that affect filename generation.

Characters that shouldn't be used in a filename

Command-Line Options: --bad-filename-chars=*string*
Config File: [files] bad-chars
Default: : #$%^&*!~‘"’=?/[]()|<>;\,.
specifies all characters that should not be allowed in a filename. These characters will be replaced by the value in **--bad-filename-chars-sub**.

String to use in place of invalid characters

Command-Line Options: --bad-filename-chars-sub=*string*
Config File: [files] bad-chars-sub
Default: -
specifies a string to use in place of invalid filename characters (specified by the **--bad-chars-sub** option)

Output Directory

Command-Line Options: --dir=*directory* `or` **-d *directory***
Config File: [files] directory
Default: $jobname
specifies a directory name to use as the output directory.

Escaping characters higher than 7-bit

Command-Line Options: --escape-high-chars

Config File: [files] escape-high-chars
Default: False
some output types allow you to represent characters that are greater than 7-bits with an alternate representation to alleviate the issue of file encoding. This option indicates that these alternate representations should be used.

Note: The renderer is responsible for doing the translation into the alternate format. This might not be supported by all output types.

Template to use for output filenames
Command-Line Options: --filename=*string*
Config File: [files] filename
specifies the templates to use for generating filenames. The filename template is a list of space separated names. Each name in the list is returned once. An example is shown below.

```
index.html toc.html file1.html file2.html
```

If you don't know how many files you are going to be reproducing, using static filenames like in the example above is not practical. For this reason, these filenames can also contain variables as described in Python's string Templates (e.g. *$title*, *$id*). These variables come from the namespace created in the renderer and include: *$id*, the ID (i.e. label) of the item, *$title*, the title of the item, and *$jobname*, the basename of the LaTeX file being processed. One special variable is *$num*. This value in generated dynamically whenever a filename with *$num* is requested. Each time a filename with *$num* is successfully generated, the value of *$num* is incremented.

The values of variables can also be modified by a format specified in parentheses after the variable. The format is simply an integer that specifies how wide of a field to create for integers (zero-padded), or, for strings, how many space separated words to limit the name to. The example below shows *$num* being padded to four places and *$title* being limited to five words.

```
sect$num(4).html $title(5).html
```

The list can also contain a wildcard filename (which should be specified last). Once a wildcard name is reached, it is used from that point on to generate the remaining filenames. The wildcard filename contains a list of alternatives to use as part of the filename indicated by a comma separated list of alternatives surrounded by a set of square brackets ([ ]). Each of the alternatives specified is tried until a filename is successfully created (i.e. all variables resolve). For example, the specification below creates three alternatives.

```
$jobname_[$id, $title, sect$num(4)].html
```

The code above is expanded to the following possibilities.

```
$jobname_$id.html
$jobname_$title.html
$jobname_sect$num(4).html
```

Each of the alternatives is attempted until one of them succeeds. In order for an alternative to succeed, all of the variables referenced in the template must be populated. For example, the *$id* variable will not be populated unless the node had a `\$label` macro pointing to it. The *title* variable would not be populated unless the node had a title associated with it (e.g. such as section, subsection, etc.). Generally, the last one should contain no variables except for *$num* as a fail-safe alternative.

Input Encoding
Command-Line Options: --input-encoding=*string*

Config File: [files] input-encoding
Default: utf-8
specifies which encoding the LaTeX source file is in

Output Encoding
Command-Line Options: --output-encoding=*string*
Config File: [files] output-encoding
Default: utf-8
specifies which encoding the output files should use. **Note:** This depends on the output format as well. While HTML and XML use encodings, a binary format like MS Word, would not.

Splitting document into multiple files
Command-Line Options: --split-level=*integer*
Config File: [files] split-level
Default: 2
specifies the highest section level that generates a new file. Each section in a LaTeX document has a number associated with its hierarchical level. These levels are -2 for the document, -1 for parts, 0 for chapters, 1 for sections, 2 for subsections, 3 for subsubsections, 4 for paragraphs, and 5 for subparagraphs. A new file will be generated for every section in the hierarchy with a value less than or equal to the value of this option. This means that for the value of 2, files will be generated for the document, parts, chapters, sections, and subsections.

2.1.6 Image Options

Images are created by renderers when the output type in incapable of rendering the content in any other way. This method was commonly used to display equations in XHTML output. Nowadays, MathJax arguably provides a better method, see Section 2.1.7 below. The following options control how images are generated.

Base URL
Command-Line Options: --image-base-url=*url*
Config File: [images] base-url
specifies a base URL to prepend to the path of all images.

LaTeX program to use to compile image document
Command-Line Options: --image-compiler=*program*
Config File: [images] compiler
Default: latex
specifies which program to use to compile the images LaTeX document.

Enable or disable image generation
Command-Line Options: --enable-images `or` **--disable-images**
Config File: [images] enabled
Default: yes
indicates whether or not images should be generated.

Enable or disable the image cache
Command-Line Options: --enable-image-cache `or` **--disable-image-cache**
Config File: [images] cache
Default: yes
indicates whether or not images should use a cache between runs.

Convert LaTeX output to images
Command-Line Options: --imager=*program*

Config File: [images] imager
Default: dvipng dvi2bitmap gsdvipng gspdfpng OSXCoreGraphics
specifies which converter will be used to take the output from the LaTeX compiler and convert it to images. You can specify a space delimited list of names as well. If a list of names is specified, each one is verified in order to see if it works on the current machine. The first one that succeeds is used.

You can use the value of "none" to turn the imager off.

Image filenames
Command-Line Options: --image-filenames=*filename-template*
Config File: [images] filenames
Default: images/img-$num(4).png
specifies the image naming template to use to generate filenames. This template is the same as the templates used by the **--filename** option.

Convert LaTeX output to vector images
Command-Line Options: --vector-imager=*program*
Config File: [images] vector-imager
Default: dvisvgm
specifies which converter will be used to take the output from the LaTeX compiler and convert it to vector images. You can specify a space delimited list of names as well. If a list of names is specified, each one is verified in order to see if it works on the current machine. The first one that succeeds is used.

You can use the value of "none" to turn the vector imager off.

Note: When using the vector imager, a bitmap image is also created using the regular imager. This bitmap is used to determine the depth information about the vector image and can also be used as a backup if the vector image is not supported by the viewer.

2.1.7 HTML5 Renderer Options

Each renderer can define its own configuration options. This section describes options from the HTML5 renderer. These options have no effect if another renderer is used. Also these options may have no effect if the default theme is not used.

The first three options give control on navigation helpers (tables of contents and breadcrumbs links). Together with the extra-css option, which allows to set css rules overriding the default ones, they allow radical changes to the output style without modifying any template or python code. See Section 5.5 for more information on the HTML5 renderer and how to customize its output.

Display table of contents on each page
Command-Line Options: --display-toc `or` **--no-display-toc**
Config File: [html5] display-toc
Default: true
specifies whether to display the table of contents on each page.

Local table of contents level
Command-Line Options: --localtoc-level=*level*
Config File: [html5] localtoc-level
Default: Node.DOCUMENT_LEVEL-1
specifies from which level one creates local table of contents. The default value implies local table of contents are never created.

Create breadcrumbs from this level
Command-Line Options: --breadcrumbs-level=*level*

Config File: [html5] breadcrumbs-level
Default: -10
specifies from which level one creates breadcrumbs navigation links.

Use theme CSS
Command-Line Options: --use-theme-css `or` **--no-theme-css**
Config File: [html5] use-theme-css
Default: True
specifies whether to use CSS files from the theme.

Theme CSS file
Command-Line Options: --theme-css=*theme*
Config File: [html5] theme-css
Default: green
specifies when CSS theme to use. Possible values are currently blue or green.

Extra CSS file
Command-Line Options: --extra-css=*filename1, ...*
Config File: [html5] extra-css
Default: ”
specifies a comma separated list of css files to use in addition the theme css. These files are copied to the output directory by the renderer and loaded by the main layout template in the list order after the theme css files (if any) and the packages css files (if any).

Use theme javascript
Command-Line Options: --use-theme-js `or` **--no-theme-js**
Config File: [html5] use-theme-js
Default: True
specifies whether to use javascript files from the theme. The default theme javascript is used to hide or show part of the table of contents and proofs.

Extra javascript
Command-Line Options: --extra-js=*filename1, ...*
Config File: [html5] extra-css
Default: ”
specifies a comma separated list of javascript files to use (in addition to those coming from the theme is the use-theme-js option is set to true). These files are copied to the output directory by the renderer and loaded by the main layout template in the list order after the theme javascript files (if any) and the packages javascript files (if any).

Use MathJax
Command-Line Options: --use-mathjax `or` **--no-mathjax**
Config File: [html5] use-mathjax
Default: True
specifies whether to use MathJax for mathematics rendering. Setting this to False only makes sense if the document contains no mathematics or if some filter is expected to handle mathematics (see **--filters** option below).

MathJax library url
Command-Line Options: --mathjax-url=*url*
Config File: [html5] mathjax-url
Default: http://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS_CHTML
specifies where to find the MathJax javascript lib (including the config information as in the default value).

Use single dollars as math delimiter for MathJax

Command-Line Options: --dollars `or` **--no-dollars**
Config File: [html5] mathjax-dollars
Default: False
specifies whether single dollars are used as math delimiters instead of `\(` and `\)`. This information is used by MathJax.

Filters applied on output

Command-Line Options: --filters=*filter1, ...*
Config File: [html5] filters
Default: ”
specifies a comma separated list of commands to invoke on each output page. Each command should expect one file to convert on stdin and output the converted file on stdout.

CHAPTER

THREE

The plasTEX Document

The plasTEX document is very similar to an XML DOM structure. In fact, you can use XML DOM methods to create and populate nodes, delete or move nodes, etc. The biggest difference between the plasTEX document and an XML document is that in XML the attributes of an element are simply string values, whereas attributes in a plasTEX document are generally document fragments that contain the arguments of a macro. Attributes can be canfigured to hold other Python objects like lists, dictionaries, and strings as well (see the section 4 for more information).

While XML document objects have a very strict syntax, LATEX documents are a little more free-form. Because of this, the plasTEX framework does a lot of normalizing of the LATEX document to make it conform to a set of rules. This set of rules means that you will always get a consistent output document which is necessary for easy manipulation and programability.

The overall document structure should not be surprising. There is a document element at the top level which corresponds to the XML Document node. The child nodes of the Document node begin with the preamble to the LATEX document. This includes things like the `\documentclass`, `\newcommand`s, `\title`, `\author`, counter settings, etc. For the most part, these nodes can be ignored. While they are a useful part of the document, they are generally only used by internal processes in plasTEX. What is important is the last node in the document which corresponds to LATEX's `document` environment.

The `document` environment has a very simple structure. It consists solely of paragraphs (actually `\par`s in TEX's terms) and sections[1]. In fact, all sections have this same format including parts, chapters, sections, subsections, subsubsections, paragraphs, and subparagraphs. plasTEX can tell which pieces of a document correspond to a sectioning element by looking at the `level` attribute of the Python class that corresponds to the given macro. The section levels in plasTEX are the same as those used by LATEX: -1 for part, 0 for chapter, 1 for section, etc. You can create your own sectioning commands simply by subclassing an existing macro class, or by setting the `level` attribute to a value that corresponds to the level of section you want to mimic. All level values less than 100 are reserved for sectioning so you aren't limited to LATEX's sectioning depth. Figure 3.1 below shows an example of the overall document structure.

This document is constructed during the parsing process by calling the `digest` method on each node. The `digest` method is passed an iterator of document nodes that correspond to the nodes in the document that follow the current node. It is the responsibility of the current node to only absorb the nodes that belong to it during the digest process. Luckily, the default `digest` method will work in nearly all cases. See section 4 for more information on the digestion process.

Part of this digestion process is grouping nodes into paragraphs. This is done using the `paragraphs` method available in all `Macro` based classes. This method uses the same technique as TEX to group paragraphs of content. Section 3.2 has more information about the details of paragraph grouping.

In addition to the `level` attribute of sections, there is also a mixin class that assists in generating the table of contents and navigation elements during rendering. If you create your own sectioning commands, you should include `plasTeX.Base.LaTeX.Sectioning.SectionUtils` as a base class as well. All of the standard LATEX section commands already inherit from this class, so if you subclass one of those, you'll get the helper methods for free. For more information on these helper methods see section 3.1.

[1] "sections" in this document is used loosely to mean any type of section: part, chapter, section, etc.

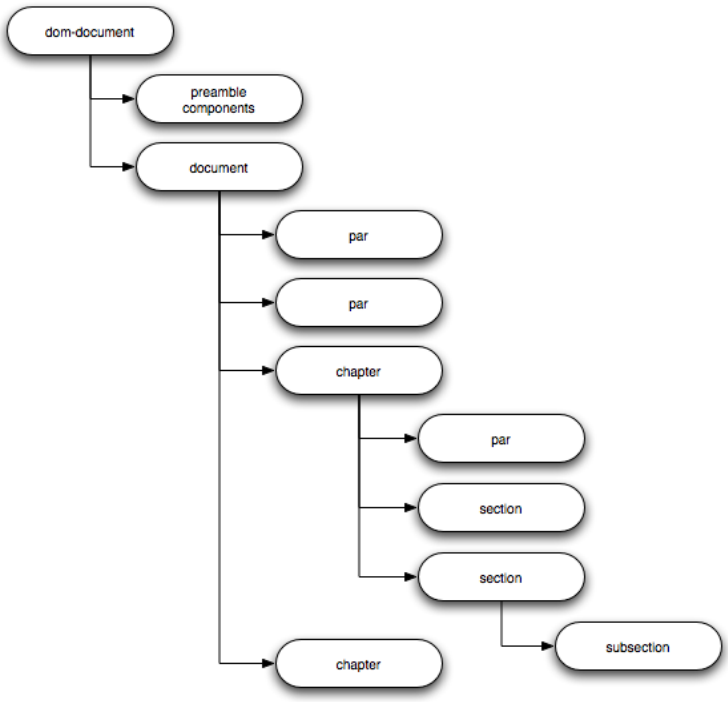

Figure 3.1: The overall plasTEX document structure

The structure of the rest of the document is also fairly simple and well-defined. LATEX commands are each converted into a document node with it's arguments getting placed into the `attributes` dictionary. LATEX environments also create a single node in the document, where the child nodes of the environment include everything between the `\begin` and `\end` commands. By default, the child nodes of an environment are simply inserted in the order that they appear in the document. However, there are some environments that require further processing due to their more complex structures. These structures include arrays and tabular environments, as well as itemized lists. For more information on these structures see sections 3.3.3 and 3.3.1, respectively. Figures 3.2 and 3.3 shows a common LATEX document fragment and the resulting plasTEX document node structure.

```
\begin{center}
Every \textbf{good} boy does \textit{fine}.
\end{center}
```

Figure 3.2: Sample LATEX document fragment code

You may have noticed that in the document structure in Figure 3.3 the text corresponding to the argument for `\textbf` and `\textit` is actually a child node and not an attribute. This is actually a convenience feature in plasTEX. For macros like this where there is only one argument and that argument corresponds to the content of the macro, it is common to put that content into the child nodes. This is done in the `args` attribute of the macro class by setting the argument's name to "self". This magical value will link the attribute called "self" to the child nodes array. For more information on the `args` attribute and how it populates the `attributes` dictionary see section 4.

In the plasTEX framework, the input LATEX document is parsed and digested until the document is finished. At this point, you should have an output document that conforms to the rules described above. The document should have a regular enough structure that working with it programatically using DOM methods or Python practices should be fairly straight-forward. The following sections give more detail on document structure elements that require extra

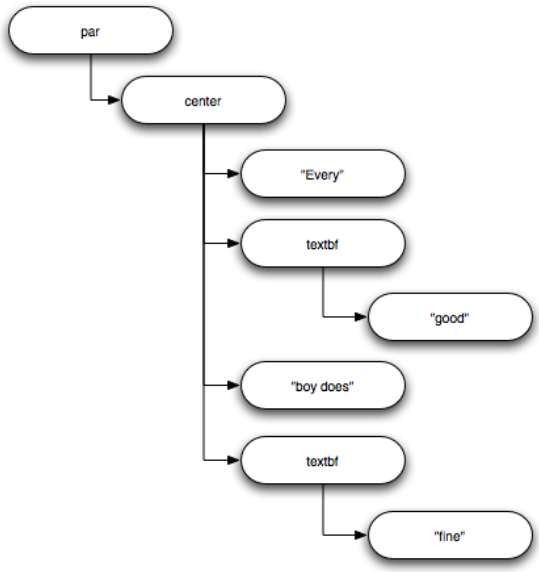

Figure 3.3: Resulting plasTEX document node structure

processing beyond the standard parse-digest process.

3.1 Sections

"Sections" in plasTEX refer to any macro that creates a section-like construct in a document including the `document` environment, `\part`, `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\paragraph`, and `\subparagraph`. While these are the sectioning macros defined by LATEX, you are not limited to using just those commands to create sections in your own documents. There are two elements that must exist for a Python macro class to act like a section: 1) the `level` attribute must be set to a value less than 100, and 2) the class should inherit from `plasTeX.Base.LaTeX.Sectioning.SectionUtils`.

The `level` attribute refers to the section level in the document. The values for this attribute are the same values that LATEX uses for its section levels, namely:

-1 *or* **`Node.PART_LEVEL`** corresponds to `\part`

0 *or* **`Node.CHAPTER_LEVEL`** corresponds to `\chapter`

1 *or* **`Node.SECTION_LEVEL`** corresponds to `\section`

2 *or* **`Node.SUBSECTION_LEVEL`** corresponds to `\subsection`

3 *or* **`Node.SUBSUBSECTION_LEVEL`** corresponds to `\subsubsection`

4 *or* **`Node.PARAGRAPH_LEVEL`** corresponds to `\paragraph`

5 *or* **`Node.SUBPARAGRAPH_LEVEL`** corresponds to `\subparagraph`

plasTEX adds the following section related levels:

`-sys.maxint` *or* **`Node.DOCUMENT_LEVEL`** corresponds to the `document` environment and is always the top-level section

6 *or* **`Node.SUBSUBPARAGRAPH_LEVEL`** this level was added to correspond to the sixth level of headings defined in HTML

100 *or* **`Node.ENDSECTIONS_LEVEL`** flag that indicates the last possible section nesting level. This is mainly used for internal purposes.

plasTEX uses the `level` attribute to build the appropriate document structure. If all you need is a proper document structure, the `level` attribute is the only thing that needs to be set on a macro. However, there are many convenience properties in the `plasTeX.Base.LaTeX.Sectioning.SectionUtils` class that are used in the rendering process. If you plan on rendering your document, your section classes should inherit from this class. Below is a list of the additional properties and their purpose.

Name	Purpose
`allSections`	contains a sequential list of all of the sections within and including the current section
`documentSections`	contains a sequential list of all of the sections within the entire document
`links`	contains a dictionary contain various amounts of navigation information corresponding mostly to the link types described at `http://fantasai.tripod.com/qref/Appendix/LinkTypes/ltdef.html`. This includes things like breadcrumb trails, previous and next links, links to the overall table of contents, etc. See section 3.1.1 for more information.
`siblings`	contains a list of all of the sibling sections
`subsections`	contains a list of all of the sections within the current section
`tableofcontents`	contains an object that corresponds to the table of contents for the section. The table of contents is configurable as well. For more information on how to configure the table of contents see section 3.1.2

Note: When first accessed, each of these properties actually navigates the document and builds the returned object. Since these operations can be rather costly, the values are cached. Therefore, if you modify the document after accessing one of these properties you will not see the change reflected.

3.1.1 Navigation and Links

The `plasTeX.Base.LaTeX.Sectioning.SectionUtils` class has a property named `links` that contains a dictionary of many useful objects that assist in creating navigation bars and breadcrumb trails in the rendered output. This dictionary was modeled after the links described at `http://fantasai.tripod.com/qref/Appendix/LinkTypes/ltdef.html`. Some of the objects in this dictionary are created automatically, others are created with the help of the `linkType` attribute on the document nodes, and yet others can be added manually from a configuration file or command-line options. The automatically generated values are listed in the following table.

Name	Purpose
begin	the first section of the document
breadcrumbs	a list containing the entire parentage of the current section (including the current section)
chapter	the current chapter node
child	a list of the subsections
contents	the section that contains the top-level table of contents
document	the document level node
end	the last section of the document
first	the first section of the document
home	the first section of the document
home	the first section of the document
last	the last section of the document
navigator	the section that contains the top-level table of contents
next	the next section in the document
origin	the section that contains the top-level table of contents
parent	the parent node
part	the current part node
prev	the previous section in the document
previous	the previous section in the document
section	the current section
sibling	a list of the section siblings
subsection	the current subsection
start	the first section of the document
toc	the section that contains the top-level table of contents
top	the first section of the document
up	the parent section

Note: The keys in every case are simply strings. **Note:** Each of the elements in the table above is either a section node or a list of section nodes. Of course, once you have a reference to a node you can acces the attributes and methods of that object for further introspection. An example of accessing these objects from a section instance is shown below.

```
previousnode = sectionnode.links['prev']
nextnode = sectionnode.links['next']
```

The next method of populating the links table is semi-automatic and uses the `linkType` attribute on the Python macro class. There are certain parts of a document that only occur once such as an index, glossary, or bibliography. You can set the `linkType` attribute on the Python macro class to a string that corresponds to that sections role in the document (i.e. 'index' for the index, 'glossary' for the glossary, 'bibliography' for the bibliography). When a node with a special link type is created, it is inserted into the dictionary of links with the given name. This allows you to have links to indexes, glossaries, etc. appear in the links object only when they are in the current document. The example below shows the `theindex` environment being configured to show up under the 'index' key in the links dictionary.

```
class theindex(Environment, SectionUtils):
    nodeType = 'index'
    level = Environment.SECTION_LEVEL
```

Note: These links are actually stored under the 'links' key of the owner document's userdata dictionary (i.e. `self.ownerDocument.userdata['links']`). Other objects can be added to this dictionary manually.

The final way of getting objects into the links dictionary is through a configuration file or command-line options. This method is described fully in section 2.1.4.

3.1.2 Table of Contents

The table of contents object returned by the `tableofcontents` property of `SectionUtils` is not an actual node of the document, but it is a proxy object that limits the number of levels that you can traverse. The number of levels that you are allowed to traverse is determined by `document:toc-depth` section of the configuration (see section 2.1.2). Other than the fact that you can only see a certain number of levels of subsections, the object otherwise acts just like any other section node.

In addition to limiting the number of levels of a table of contents, you can also determine whether or not sections that do not generate new files while rendering should appear in the table of contents. By default, only sections that generate a new file while rendering will appear in the table of contents object. If you set the value of `document:toc-non-files` in the configuration to `True`, then all sections will appear in the table of contents.

3.2 Paragraphs

Paragraphs in a plasTEX document are grouped in the same way that they are grouped in TEX: essentially anything within a section that isn't a section itself is put into a paragraph. This is different than the HTML model where tables and lists are not grouped into paragraphs. Because of this, it is likely that HTML generated that keeps the same paragraph model will not be 100% valid. However, it is highly unlikely that this variance from validity will cause any real problems in the browser rendering the correct output.

Paragraphs are grouped using the `paragraphs` method available on all Python macro classes. When this method is invoked on a node, all of the child nodes are grouped into paragraphs. If there are no paragraph objects in the list of child nodes already, one is created. This is done to make sure that the document is fully normalized and that paragraphs occur everywhere that they can occur. This is most noteworthy in constructs like tables and lists where some table cells or list items have multiple paragraphs and others do not. If a paragraph weren't forced into these areas, you could have inconsistently paragraph-ed content.

Some areas where paragraphs are allowed, but not necessarily needed might not want the forced paragraph to be generated, such as within a grouping of curly braces ({ }). In these cases, you can use the `force=False` keyword argument to `paragraphs`. This still does paragraph grouping, but only if there is a paragraph element already in the list of child nodes.

3.3 Complex Structures

While much of a plasTEX document mirrors the structure of the source LATEX document, some constructs do require a little more work to be useful in the more rigid structure. The most noteworthy of these constructs are lists, arrays (or tabular environments), and indexes. These objects are described in more detail in the following sections.

3.3.1 Lists

Lists are normalized slightly more than the rest of the document. They are treated almost like sections in that they are only allowed to contain a minimal set of child node types. In fact, lists can only contain one type of child node: list item. The consequence of this is that any content before the first item in a list will be thrown out. In turn, list items will only contain paragraph nodes. The structure of all list structures will look like the structure in Figure 3.4.

This structure allows you to easily traverse a list with code like the following.

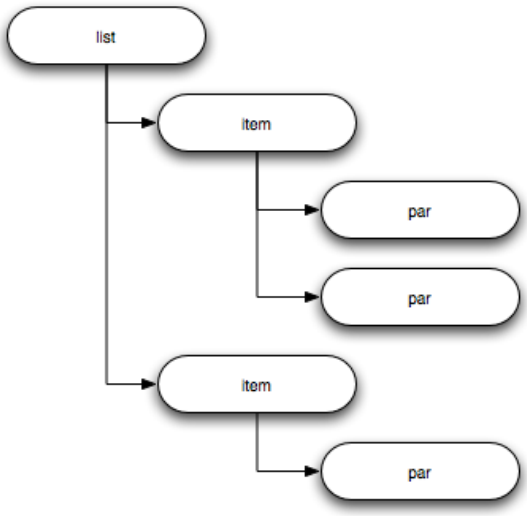

Figure 3.4: Normalized structure of all lists

```
# Iterate through the items in the list node
for item in listnode:

    # Iterate through the paragraphs in each item
    for par in item:

        # Print the text content of each paragraph
        print par.textContent

    # Print a blank line to separate each item
    print
```

3.3.2 Bibliography

The bibliography is really just another list structure with a few enhancements to allow referencing of the items throughout the document. Bibliography processing is left to the normal tools. plasTEX expects a properly '.bbl' file for the bibliography. The LATEX bibliography is the format used by default; however, the natbib package is also included with plasTEX for more complex formatting of bibliographies.

3.3.3 Arrays and Tabular Environments

Arrays and tabular environments are the most complex structures in a plasTEX document. This because tables can include spanning columns, spanning rows, and borders specified on the table, rows, and individual cells. In addition, there are alignments associated with each column and alignments can be specified by any `\multicolumn` command. It is also possible with some packages to create your own column declarations. Add to that the fact that the longtable package allows you to specify multiple headers, footers, and coptions, and you can see why tabular environments can be rather tricky to deal with.

As with all parts of the document, plasTEX tries to normalize all tables to have a consistent structure. The structure for

arrays and tables is shown in Figure 3.5.

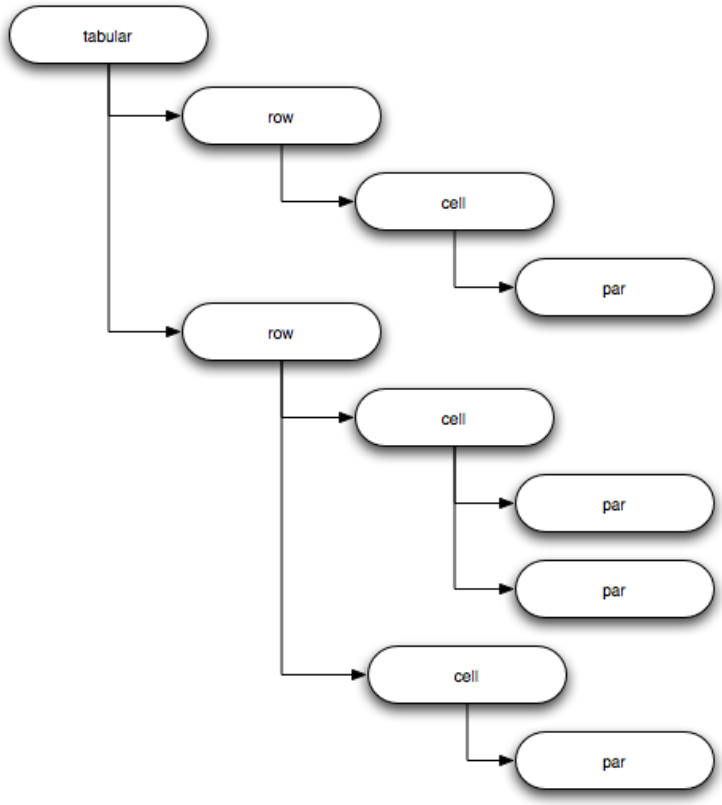

Figure 3.5: Normalized structure of all tables and arrays

Luckily, the array macro class that comes with plasTEX was made to handle all of the work for you. In fact, it also handles the work of some extra packages such as longtable to make processing them transparent. The details of the tabular environments are described in the following sections.

With this normalized structure, you can traverse all array and table structures with code like the following.

```
# Iterate through all rows in the table
for row in tablenode:

    # Iterate through all cells in the row
    for cell in row:

        # Iterate through all paragraphs in the cell
        for par in cell:

            # Print the text content of each cell
            print '   ' + par.textContent

        # Print a blank line after each cell
        print

    # Print a blank line after each row
    print
```

Borders

Borders in a tabular environment are generally handled by `\hline`, `\vline`, `\cline`, as well as the column specifications on the tabular environment and the `\multicolumn` command. plasTEX merges all of the border specifications and puts them into CSS formatted values in the `style` attribute of each of the table cell nodes. To get the CSS information formatted such that it can be used in an inline style, simply access the `inline` property of the style object.

Here is an example of a `tabular` environment.

```
\begin{tabular}{|l|l|}\hline
x & y \\
1 & 2 \\\hline
\end{tabular}
```

The table node can be traversed as follows.

```
# Print the CSS for the borders of each cell
for rownum, row in enumerate(table):
    for cellnum, cell in enumerate(row):
        print '(%s,%s) %s -- %s' % (rownum, cellnum,
               cell.textContent.strip(), cell.style.inline)
```

The code above will print the following output (whitespace has been added to make the output easier to read).

```
(0,0) x -- border-top-style:solid;
           border-left:1px solid black;
           border-right:1px solid black;
           border-top-color:black;
           border-top-width:1px;
           text-align:left
(0,1) y -- border-top-style:solid;
           text-align:left;
           border-top-color:black;
           border-top-width:1px;
           border-right:1px solid black
(1,0) 1 -- border-bottom-style:solid;
           border-bottom-width:1px;
           border-left:1px solid black;
           border-right:1px solid black;
           text-align:left;
           border-bottom-color:black
(1,1) 2 -- border-bottom-color:black;
           border-bottom-width:1px;
           text-align:left;
           border-bottom-style:solid;
           border-right:1px solid black
```

Alignments

Alignments can be specified in the column specification of the tabular environment as well as in the column specification of \multicolumn commands. Just like the border information, the alignment information is also stored in CSS formatted values in each cell's `style` attribute.

Longtables

Longtables are treated just like regular tables. Only the first header and the last footer are supported in the resulting table structure. To indicate that these are verifiable header or footer cells, the `isHeader` attribute of the corresponding cells is set to `True`. This information can be used by the renderer to more accurately represent the table cells.

3.3.4 Indexes

All index building and sorting is done internally in plasTEX. It is done this way because the information that tools like **makeindex** generate is only useful to LATEX itself since the refence to the place where the index tag was inserted is simply a page number. Since plasTEX wants to be able to reference the index tag node, it has to do all of the index processing natively.

There are actually two index structures. The default structure is simply the index nodes sorted and grouped into the appropriate hierarchies. This structure looks like the structure pictured in Figure 3.6.

Each item, subitem, and subsubitem has an attribute called `key` that contains a document fragment of the key for that index item. The document nodes that this key corresponds to are held in a list in the `pages` attribute. These nodes are the actual nodes corresponding to the index entry macros from the LATEX document. The content of the node is a number corresponding to the index entry that is formatted according to the formatting rules specified in the index entry.

While the structure above works well for paged media, it is sometimes nice to have the index entries grouped by first letter and possibly even arranged into multiple columns. This alternate representation can be accessed in the `groups`

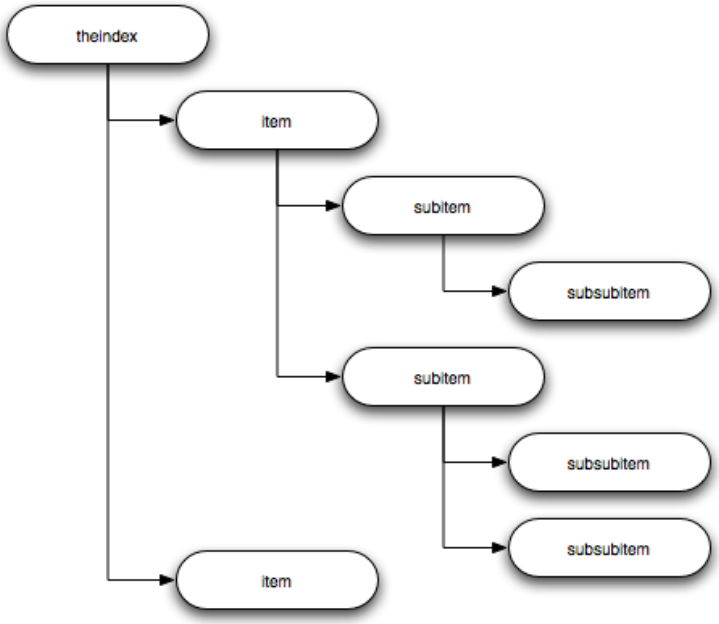

Figure 3.6: Default index structure

property. The structure for this type of index is shown in Figure 3.7.

In this case, the item, subitem, and subsubitem nodes are the same as in the default scheme. The group has a `title` attribute that contains the first letter of the entries in that group. Entries that start with something other than a letter or an underscore are put into a group called “Symbols”. The columns are approximately equally sized columns of index entries. The number of columns is determined by the `document:index-columns` configuration item.

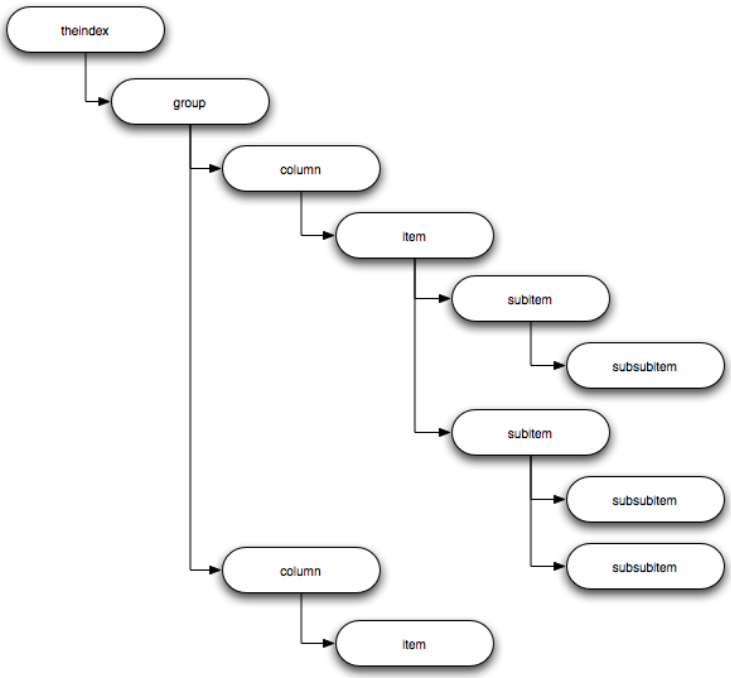

Figure 3.7: Grouped index structure

CHAPTER

FOUR

Understanding Macros and Packages

Macros and packages in plasTEX live a dual life. On one hand, macros can be defined in LATEX files and expanded by plasTEX itself. On the other hand, macros can also be implemented as Python classes. Packages are the same way. plasTEX can handle some LATEX packages natively. Others may have to be implemented in Python. In most cases, both implementations work transparently together. If you don't define that many macros, and the ones that you do define are simple or even of intermediate complexity, it's probably better to just let plasTEX handle them natively. However, there are some reasons that you may want to implement Python versions of your macros:

- Python versions of macros are generally faster
- You have more control over what gets inserted into the output document
- You can store information in the document's `userdata` dictionary for use later
- You can prevent a macro from being expanded into primitive LATEX commands, so that a custom renderer can be used on that node
- Some macros just don't make sense in a plasTEX document
- Some macros are just too complicated for plasTEX

If any of these reasons appeal to you, read the following sections on how to implement macros and packages in plasTEX.

4.1 Defining Macros in LATEX

Defining macros in LATEX using plasTEX is no different than the way you would normally define you macros; however, there is a trick that you can use to improve you macros for plasTEX, if needed. While plasTEX can handle fairly complicated macros, some macros might do things that don't make sense in the context of a plasTEX document, or they might just be too complicated for the plasTEX engine to handle. In cases such as these, you can use the `\ifplastex` construct. As you may know in TEX, you can define your own `\if` commands using the `\newif` primitive. There is an `\if` command called `\ifplastex` built into the plasTEX engine that is always set to true. In you document, you can define this command and set it to false (as far as LATEX is concerned) as follows.

```
\newif\ifplastex
\plastexfalse
```

Now you can surround the portions of your macros that plasTEX has trouble with, or even write alternative versions of the macro for LATEX and plasTEX. Here is an example.

```
\newcommand{\foo}[1]{
    \ifplastex\else\vspace*{0.25in}\fi
    \textbf{\Large{#1}}
    \ifplastex\else\vspace*{1in}\fi
}

\ifplastex
    \newenvironment{coolbox}{}{}
\else
    \newenvironment{coolbox}
        {fbox\bgroup\begin{minipage}{5in}}
        {\end{minipage}\egroup}
\fi
```

4.2 Defining Macros in Python

Defining macros using Python classes (or, at least through Python interfaces) is done in one of three ways: INI files, Python classes, and the document context. These three methods are described in the following sections.

4.2.1 Python Classes

Both LaTeX command and environments can be implemented in Python classes. plasTeX includes a base class for each one: `Command` for commands and `Environment` for environments. For the most part, these two classes behave in the same way. They both are responsible for parsing their arguments, organizing their child nodes, incrementing counters, etc. much like their LaTeX counterparts. The Python macro class feature set is based on common LaTeX conventions. So if the LaTeX macro you are implementing in Python uses standard LaTeX conventions, you job will be very easy. If you are doing unconventional operations, you will probably still succeed, you just might have to do a little more work.

The three most important parts of the Python macro API are: 1) the `args` attribute, 2) the `invoke` method, and 3) the `digest` method. When writing your own macros, these are used the most by far.

The `args` Attribute

The `args` attribute is a string attribute on the class that indicates what the arguments to the macro are. In addition to simply indicating the number of arguments, whether they are mandatory or optional, and what characters surround the argument as in LaTeX, the `args` string also gives names to each of the argument and can also indicate the content of the argument (i.e. int, float, list, dictionary, string, etc.). The names given to each argument determine the key that the argument is stored under in the the `attributes` dictionary of the class instance. Below is a simple example of a macro class.

```
from plasTeX import Command, Environment

class framebox(Command):
    """ \framebox[width][pos]{text} """
    args = '[ width ] [ pos ] text'
```

In the `args` string of the `\framebox` macro, three arguments are defined. The first two are optional and the third one is mandatory. Once each argument is parsed, in is put into the `attributes` dictionary under the name given in the

`args` string. For example, the `attributes` dictionary of an instance of `\framebox` will have the keys "width", "pos", and "text" once it is parsed and can be accessed in the usual Python way.

```
self.attributes['width']
self.attributes['pos']
self.attributes['text']
```

In plasTEX, any argument that isn't mandatory (i.e. no grouping characters in the `args` string) is optional[1]. This includes arguments surrounded by parentheses (( )), square brackets ([ ]), and angle brackets (< >). This also lets you combine multiple versions of a command into one macro. For example, the `\framebox` command also has a form that looks like: `\framebox(x_dimen,y_dimen)[pos]{text}`. This leads to the Python macro class in the following code sample that encompasses both forms.

```
from plasTeX import Command, Environment

class framebox(Command):
    """

    \framebox[width][pos]{text} or
    \framebox(x_dimen,ydimen)[pos]{text}

    """
    args = '( dimens ) [ width ] [ pos ] text'
```

The only thing to keep in mind is that in the second form, the *pos* attribute is going to end up under the *width* key in the `attributes` dictionary since it is the first argument in square brackets, but this can be fixed up in the `invoke` method if needed. Also, if an optional argument is not present on the macro, the value of that argument in the `attributes` dictionary is set to *None*.

As mentioned earlier, it is also possible to convert arguments to data types other than the default (a document fragment). A list of the available types is shown in the table below.

[1] While this isn't always true when LaTeX expands the macros, it will not cause any problems when plasTEX compiles the document because plasTEX is less stringent.

Name	Purpose
str	expands all macros then sets the value of the argument in the `attributes` dictionary to the string content of the argument
chr	same as 'str'
char	same as 'str'
cs	sets the attribute to an unexpanded control sequence
label	expands all macros, converts the result to a string, then sets the current label to the object that is in the `currentlabel` attribute of the document context. Generally, an object is put into the `currentlabel` attribute if it incremented a counter when it was invoked. The value stored in the `attributes` dictionary is the string value of the argument.
id	same as 'label'
idref	expands all macros, converts the result to a string, retrieves the object that was labeled by that value, then adds the labeled object to the `idref` dictionary under the name of the argument. This type of argument is used in commands like `\ref` that must reference other abjects. The nice thing about 'idref' is that it gives you a reference to the object itself which you can then use to retrieve any type of information from it such as the reference value, title, etc. The value stored in the `attributes` dictionary is the string value of the argument.
ref	same as 'idref'
nox	just parses the argument, but doesn't expand the macros
list	converts the argument to a Python list. By default, the list item separator is a comma (,). You can change the item separator in the args string by appending a set of parentheses surrounding the separator character immediately after 'list'. For example, to specify a semi-colon separated list for an argument called "foo" you would use the `args` string: "foo:list(;)". It is also possible to cast the type of each item by appending another colon and the data type from this table that you want each item to be. However, you are limited to one data type for every item in the list.
dict	converts the argument to a Python dictionary. This is commonly used by arguments set up using LaTeX's 'keyval' package. By default, key/value pairs are separated by commas, although this character can be changed in the same way as the delimiter in the 'list' type. You can also cast each value of the dictionary using the same method as the 'list' type. In all cases, keys are converted to strings.
dimen	reads a dimension and returns an instance of `dimen`
dimension	same as 'dimen'
length	same as 'dimen'
number	reads an integer and returns a Python integer
count	same as 'number'
int	same as 'number'
float	reads a decimal value and returns a Python float
double	same as 'float'

There are also several argument types used for more low-level routines. These don't parse the typical LaTeX arguments, they are used for the somewhat more free-form TeX arguments.

Name	Purpose
Dimen	reads a TEX dimension and returns an instance of `dimen`
Length	same as 'Dimen'
Dimension	same as 'Dimen'
MuDimen	reads a TEX mu-dimension and returns an instance of `mudimen`
MuLength	same as 'MuDimen'
Glue	reads a TEX glue parameter and returns an instance of `glue`
Skip	same as 'MuLength'
Number	reads a TEX integer parameter and returns a Python integer
Int	same as 'Number'
Integer	same as 'Number'
Token	reads an unexpanded token
Tok	same as 'Token'
XToken	reads an expanded token
XTok	same as 'XToken'
Args	reads tokens up to the first begin group (i.e. {)

To use one of the data types, simple append a colon (:) and the data type name to the attribute name in the `args` string. Going back to the `\framebox` example, the argument in parentheses would be better represented as a list of dimensions. The *width* parameter is also a dimension, and the *pos* parameter is a string.

```
from plasTeX import Command, Environment

class framebox(Command):
    """

    \framebox[width][pos]{text} or
    \framebox(x_dimen,ydimen)[pos]{text}

    """
    args = '( dimens:list:dimen ) [ width:dimen ] [ pos:chr ] text'
```

The `invoke` Method

The `invoke` method is responsible for creating a new document context, parsing the macro arguments, and incrementing counters. In most cases, the default implementation will work just fine, but you may want to do some extra processing of the macro arguments or counters before letting the parsing of the document proceed. There are actually several methods in the API that are called within the scope of the `invoke` method: `preParse`, `preArgument`, `postArgument`, and `postParse`.

The order of execution is quite simple. Before any arguments have been parsed, the `preParse` method is called. The `preArgument` and `postArgument` methods are called before and after each argument, respectively. Then, after all arguments have been parsed, the `postParse` method is called. The default implementations of these methods handle the stepping of counters and setting the current labeled item in the document. By default, macros that have been "starred" (i.e. have a '*' before the arguments) do not increment the counter. You can override this behavior in one of these methods if you prefer.

The most common reason for overriding the `invoke` method is to post-process the arguments in the `attributes` dictionary, or add information to the instance. For example, the `\color` command in LATEX's color package could convert the LATEX color to the correct CSS format and add it to the CSS style object.

```
from plasTeX import Command, Environment

def latex2htmlcolor(arg):
    if ',' in arg:
        red, green, blue = [float(x) for x in arg.split(',')]
        red = min(int(red * 255), 255)
        green = min(int(green * 255), 255)
        blue = min(int(blue * 255), 255)
    else:
        try:
            red = green = blue = float(arg)
        except ValueError:
            return arg.strip()
    return '#%.2X%.2X%.2X' % (red, green, blue)

class color(Environment):
    args = 'color:str'
    def invoke(self, tex):
        a = Environment.invoke(tex)
        self.style['color'] = latex2htmlcolor(a['color'])
```

While simple things like attribute post-processing is the most common use of the `invoke` method, you can do very advanced things like changing category codes, and iterating over the tokens in the TEX processor directly like the `verbatim` environment does.

One other feature of the `invoke` method that may be of interest is the return value. Most `invoke` method implementations do not return anything (or return *None*). In this case, the macro instance itself is sent to the output stream. However, you can also return a list of tokens. If a list of tokens is returned, instead of the macro instance, those tokens are inserted into the output stream. This is useful if you don't want the macro instance to be part of the output stream or document. In this case, you can simply return an empty list.

The `digest` Method

The `digest` method is responsible for converting the output stream into the final document structure. For commands, this generally doesn't mean anything since they just consist of arguments which have already been parsed. Environments, on the other hand, have a beginning and an ending which surround tokens that belong to that environment. In most cases, the tokens between the `\begin` and `\end` need to be absorbed into the `childNodes` list.

The default implementation of the `digest` method should work for most macros, but there are instances where you may want to do some extra processing on the document structure. For example, the `\caption` command within `figure`s and `table`s uses the `digest` method to populate the enclosing figure/table's `caption` attribute.

```
from plasTeX import Command, Environment

class Caption(Command):
    args = '[ toc ] self'

    def digest(self, tokens):
        res = Command.digest(self, tokens)

        # Look for the figure environment that we belong to
        node = self.parentNode
        while node is not None and not isinstance(node, figure):
            node = node.parentNode

        # If the figure was found, populate the caption attribute
        if isinstance(node, figure):
            node.caption = self

        return res

class figure(Environment):
    args = '[ loc:str ]'
    caption = None
    class caption_(Caption):
        macroName = 'caption'
        counter = 'figure'
```

More advanced uses of the `digest` method might be to construct more complex document structures. For example, tabular and array structures in a document get converted from a simple list of tokens to complex structures with lots of style information added (see section 3.3.3). One simple example of a `digest` that does something extra is shown below. It looks for the first node with the name "item" then bails out.

```
from plasTeX import Command, Environment

class toitem(Command):
    def digest(self, tokens):
        """ Throw away everything up to the first 'item' token """
        for tok in tokens:
            if tok.nodeName == 'item':
               # Put the item back into the stream
               tokens.push(tok)
               break
```

One of the more advanced uses of the `digest` is on the sectioning commands: `\section`, `\subsection`, etc. The digest method on sections absorb tokens based on the `level` attribute which indicates the hierarchical level of the node. When digested, each section absorbs all tokens until it reaches a section that has a level that is equal to or higher than its own level. This creates the overall document structure as discussed in section 3.

Other Nifty Methods and Attributes

There are many other attributes and methods on macros that can be used to affect their behavior. For a full listing, see the API documentation in section 6.1. Below are descriptions of some of the more commonly used attributes and methods.

The `level` attribute The `level` attribute is an integer that indicates the hierarchical level of the node in the output document structure. The values of this attribute are taken from LaTeX: `\part` is -1, `\chapter` is 0, `\section` is 1, `\subsection` is 2, etc. To create your owne sectioning commands, you can either subclass one of the existing sectioning macros, or simply set its `level` attribute to the appropriate number.

The `macroName` attribute The `macroName` attribute is used when you are creating a LaTeX macro whose name is not a legal Python class name. For example, the macro `\@ifundefined` has a '@' in the name which isn't legal in a Python class name. In this case, you could define the macro as shown below.

```
class ifundefined_(Command):
    macroName = '@ifundefined'
```

The `counter` attribute The `counter` attribute associates a counter with the macro class. It is simply a string that contains the name of the counter. Each time that an instance of the macro class is invoked, the counter is incremented (unless the macro has a '*' argument).

The `ref` attribute The `ref` attribute contains the value normally returned by the `\ref` command.

The `title` attribute The `title` attribute retrieves the "title" attribute from the `attributes` dictionary. This attribute is also overridable.

The `fullTitle` attribute The same as the `title` attribute, but also includes the counter value at the beginning.

The `tocEntry` attribute The `tocEntry` attribute retrieves the "toc" attribute from the `attributes` dictionary. This attribute is also overridable.

The `fullTocEntry` attribute The same as the `tocEntry` attribute, but also includes the counter value at the beginning.

The `style` attribute The `style` attribute is a CSS style object. Essentially, this is just a dictionary where the key is the CSS property name and the value is the CSS property value. It has an attribute called `inline` which contains an inline version of the CSS properties for use in the style= attribute of HTML elements.

The `id` attribute This attribute contains a unique ID for the object. If the object was labeled by a `\label` command, the ID for the object will be that label; otherwise, an ID is generated.

The `source` attribute The `source` attribute contains the LaTeX source representation of the node and all of its contents.

The `currentSection` attribute The `currentSection` attribute contains the section that the node belongs to.

The `expand` method The `expand` method is a thin wrapper around the `invoke` method. It simply invokes the macro and returns the result of expanding all of the tokens. Unlike `invoke`, you will always get the expanded node (or nodes); you will not get a *None* return value.

The `paragraphs` method The `paragraphs` method does the final processing of paragraphs in a node's child nodes. It makes sure that all content is wrapped within paragraph nodes. This method is generally called from the `digest` method.

4.2.2 INI Files

Using INI files is the simplest way of creating customized Python macro classes. It does require a little bit of knowledge of writing macros in Python classes (section 4.2.1), but not much. The only two pieces of information about Python macro classes you need to know are 1) the `args` string format, and 2) the superclass name (in most cases, you can simply use `Command` or `Environment`). The INI file features correspond to Python macros in the following way.

INI File	**Python Macro Use**
section name	the Python class to inherit from
option name	the name of the macro to create
option value	the args string for the macro

Here is an example of an INI file that defines several macros.

```
[Command]
; \program{ self }
program=self
; \programopt{ self }
programopt=self

[Environment]
; \begin{methoddesc}[ classname ]{ name  { args } ... \end{methoddesc}
methoddesc=[ classname ] name args
; \begin{memberdesc}[ classname ]{ name  { args } ... \end{memberdesc}
memberdesc=[ classname ] name args

[section]
; \headi( options:dict )[ toc ]{ title }
headi=( options:dict ) [ toc ] title

[subsection]
; \headii( options:dict )[ toc ]{ title }
headii=( options:dict ) [ toc ] title
```

In the INI file above, six macro are being defined. `\program` and `\programopt` both inherit from `Command`, the generic LaTeX macro superclass. They also both take a single mandatory argument called "self." There are two environments defined also: `methoddesc` and `memberdesc`. Each of these has three arguments where the first argument is optional. The last two macros actually inherit from standard LaTeX sectioning commands. They add an option, surrounded by parentheses, to the options that `\section` and `\subsection` already had defined.

INI versions of plasTeX packages are loaded much in the same way as Python plasTeX packages. For details on how packages are loaded, see section 4.3.

4.2.3 The Document Context

It is possible to define commands using the same interface that is used by the plasTeX engine itself. This interface belongs to the `Context` object (usually accessed through the document object's `context` attribute). Defining

commands using the context object is generally done in the `ProcessOptions` function of a package. The following methods of the context object create new commands.

Method	Purpose
`newcounter`	creates a new counter, and also creates a command called `\the`*counter* which generates the formatted version of the counter. This macro corresponds to the `\newcounter` macro in LaTeX.
`newcount`	corresponds to TeX's `\newcount` command.
`newdimen`	corresponds to TeX's `\newdimen` command.
`newskip`	corresponds to TeX's `\newskip` command.
`newmuskip`	corresponds to TeX's `\newmuskip` command.
`newif`	corresponds to TeX's `\newif` command. This command also generates macros for `\`*if command*`true` and `\`*if command*`false`.
`newcommand`	corresponds to LaTeX's `\newcommand` macro.
`newenvironment`	corresponds to LaTeX's `\newenvironment` macro.
`newdef`	corresponds to TeX's `\def` command.
`chardef`	corresponds to TeX's `\chardef` command.

Note: Since many of these methods accept strings containing LaTeX markup, you need to remember that the category codes of some characters can be changed during processing. If you are defining macros using these methods in the `ProcessOptions` function in a package, you should be safe since this function is executed in the preamble of the document where category codes are not changed frequently. However, if you define a macro with this interface in a context where the category codes are not set to the default values, you will have to adjust the markup in your macros accordingly.

Below is an example of using this interface within the context of a package to define some commands. For the full usage of these methods see the API documentation of the `Context` object in section 6.5.

```
def ProcessOptions(options, document):
    context = document.context

    # Create some counters
    context.newcounter('secnumdepth', initial=3)
    context.newcounter('tocdepth', initial=2)

    # \newcommand{\config}[2][general]{\textbf{#2:#1}
    context.newcommand('config', 2, r'\textbf{#2:#1}', opt='general')

    # \newenvironment{note}{\textbf{Note:}}{}
    context.newenvironment('note', 0, (r'\textbf{Note:}', r''))
```

4.3 Packages

Packages in plasTeX are loaded in one of three ways: standard LaTeX package, Python package, and INI file. LaTeX packages are loaded in much the same way that LaTeX itself loads packages. The **kpsewhich** program is used to locate the requested file which can be either in the search path of your LaTeX distribution or in one of the directories specified in the `TEXINPUTS` environment variable. plasTeX read the file and expand the macros therein just as LaTeX would do.

Python packages are located using Python's search path. This includes all directories listed in *sys.path* as well as those listed in the `PYTHONPATH` environment variable. After a package is loaded, it is checked to see if there is a function called `ProcessOptions` in its namespace. If there is, that function is called with two arguments: 1) the dictionary of options that were specified when loading the package, and 2) the document object that is currently being processed. This function allows you to make adjustments to the loaded macros based on the options specified, and define new

commands in the document's context (see section 4.2.3 for more information). Of course, you can also define Python based macros (section 4.2.1) in the Python package as well.

The last type of packages is based on the INI file format. This format is discussed in more detail in section 4.2.2. INI formatted packages are loaded in conjunction with a LaTeX or Python package. When a package is loaded, an INI file with the same basename is searched for in the same director as the package. If it exists, it is loaded as well. For example, if you had a package called 'python.sty' and a file called 'python.ini' in the same package directory, 'python.sty' would be loaded first, then 'python.ini' would be loaded. The same operation applies for Python based packages.

CHAPTER

FIVE

Renderers

Renderers allow you to convert a plasTEX document object into viewable output such as HTML, RTF, or PDF, or simply a data structure format such as DocBook or tBook. Since the plasTEX document object gives you everything that you could possibly want to know about the LATEX document, it should, in theory, be possible to generate any type of output from the plasTEX document object while preserving as much information as the output format is capable of. In addition, since the document object is not affected by the rendering process, you can apply multiple renderers in sequence so that the LATEX document only needs to be parsed one time for all output types.

While it is possible to write a completely custom renderer, one possible rendeerer implementation is included with the plasTEX framework. While the rendering process in this implementation is fairly simple, it is also very powerful. Some of the main features are listed below.

- ability to generate multiple output files
- automatic splitting of files is configurable by section level, or can be invoked using ad-hoc methods in the `filenameoverride` property
- powerful output filename generation utility
- image generation for portions of the document that cannot be easily rendered in a particular output formate (e.g. equations in HTML)
- themeing support
- hooks for post-processing of output files
- configurable output encodings

The API of the renderer itself is very small. In fact, there are only a couple of methods that are of real interest to an end user: `render` and `cleanup`. The `render` method is the method that starts the rendering process. It's only argument is a plasTEX document object. The `cleanup` method is called at the end of the rendering process. It is passed the document object and a list of all of the files that were generated. This method allows you to do post-processing on the output files. In general, this method will probably only be of interest to someone writing a subclass of the `Renderer` class, so most users of plasTEX will only use the `render` method. The real work of the rendering process is handled in the `Renderable` class which is discussed later in this chapter.

The `Renderer` class is a subclass of the Python dictionary. Each key in the renderer corresponds to the name of a node in the document object. The value stored under each key is a function. As each node in the document object is traversed, the renderer is queried to see if there is a key that matches the name of the node. If a key is found, the value at that key (which must be a function) is called with the node as its only argument. The return value from this call must be a unicode object that contains the rendered output. Based on the configuration, the renderer will handle all of the file generation and encoding issues.

If a node is traversed that doesn't correspond to a key in the renderer dictionary, the default rendering method is called. The default rendering method is stored in the `default` attribute. One exception to this rule is for text nodes. The

default rendering method for text nodes is actually stored in `textDefault`. Again, these attributes simply need to reference any Python function that returns a unicode object of the rendered output. The default method in both of these attributes is the `unicode` built-in function.

As mention previously, most of the work of the renderer is actually done by the `Renderable` class. This is a mixin class[1] that is mixed into the `Node` class in the `render` method. It is unmixed at the end of the `render` method. The details of the `Renderable` class are discussed in section 5.2.

5.1 Simple Renderer Example

It is possible to write a renderer with just a couple of methods: `default` and `textDefault`. The code below demonstrates how one might create a generic XML renderer that simply uses the node names as XML tag names. The text node renderer escapes the <, >, and & characters.

```
import string
from plasTeX.Renderers import Renderer

class Renderer(Renderer):

    def default(self, node):
        """ Rendering method for all non-text nodes """
        s = []

        # Handle characters like \&, \$, \%, etc.
        if len(node.nodeName) == 1 and node.nodeName not in string.letters:
            return self.textDefault(node.nodeName)

        # Start tag
        s.append('<%s>' % node.nodeName)

        # See if we have any attributes to render
        if node.hasAttributes():
            s.append('<attributes>')
            for key, value in node.attributes.items():
                # If the key is 'self', don't render it
                # these nodes are the same as the child nodes
                if key == 'self':
                    continue
                s.append('<%s>%s</%s>' % (key, unicode(value), key))
            s.append('</attributes>')

        # Invoke rendering on child nodes
        s.append(unicode(node))

        # End tag
        s.append('</%s>' % node.nodeName)

        return u'\n'.join(s)

    def textDefault(self, node):
        """ Rendering method for all text nodes """
        return node.replace('&','&amp;').replace('<','&lt;').replace('>','&gt;')
```

[1] A mixin class is simply a class that is merely a collection of methods that are intended to be included in the namespace of another class.

To use the renderer, simply parse a LaTeX document and apply the renderer using the `render` method.

```
# Import renderer from previous code sample
from MyRenderer import Renderer

from plasTeX.TeX import TeX

# Instantiate a TeX processor and parse the input text
tex = TeX()
tex.ownerDocument.config['files']['split-level'] = -100
tex.ownerDocument.config['files']['filename'] = 'test.xml'
tex.input(r'''
\documentclass{book}
\begin{document}

Previous paragraph.

\section{My Section}

\begin{center}
Centered text with <, >, and \& charaters.
\end{center}

Next paragraph.

\end{document}
''')
document = tex.parse()

# Render the document
renderer = Renderer()
renderer.render(document)
```

The output from the renderer, located in 'test.xml', looks like the following.

```
<document>
<par>
Previous paragraph.
</par><section>
    <attributes>
        <toc>None</toc>
        <*modifier*>None</*modifier*>
        <title>My Section</title>
    </attributes>
<par>
<center>
 Centered text with &lt;, &gt;, and &amp; charaters.
</center>
</par><par>
Next paragraph.
</par>
</section>
</document>
```

5.1.1 Extending the Simple Renderer

Now that we have a simple renderer working, it is very simple to extend it to do more specific operations. Let's say that the default renderer is fine for most nodes, but for the `\section` node we want to do something special. For the section node, we want the title argument to correspond to the title attribute in the output XML[2]. To do this we need a method like the following.

```
def handle_section(node):
    return u'\n\n<%s title="%s">\n%s\n</%s>\n' % \
            (node.nodeName, unicode(node.attributes['title']),
             unicode(node), node.nodeName)
```

Now we simply insert the rendering method into the renderer under the appropriate key. Remember that the key in the renderer should match the name of the node you want to render. Since the above rendering method will work for all section types, we'll insert it into the renderer for each LaTeX sectioning command.

```
renderer = Renderer()
renderer['section'] = handle_section
renderer['subsection'] = handle_section
renderer['subsubsection'] = handle_section
renderer['paragraph'] = handle_section
renderer['subparagraph'] = handle_section
renderer.render(document)
```

Running the same LaTeX document as in the previous example, we now get this output.

```
<document>
<par>
Previous paragraph.
</par>

<section title="My Section">
<par>
<center>
 Centered text with &lt;, &gt;, and &amp; charaters.
</center>
</par><par>
Next paragraph.
</par>
</section>

</document>
```

Of course, you aren't limited to using just Python methods. Any function that accepts a node as an argument can be used. The Page Template renderer included with plasTeX is an example of how to write a renderer that uses a templating language to render the nodes (see section 5.3).

[2]This will only work properly in XML if the content of the title is plain text since other nodes will generate markup.

5.1.2 Using a Renderer from the plastex Script

In the preceding sections, the simple renderer example was called from a custom python script. In order to use it through the main plastex script (described in Chapter 2), it needs to be located in some directory `plasTeX/Renderers/SimpleRenderer`, where `plasTeX` is the directory containing the plastex script. This directory must contain a `__init__.py` file defining the *Renderer* class (with this name). This directory can also contain a `Themes` directory in order to use the theme option described in Section 2.1.1. Each subdirectory in the `Themes` directory is considered as a theme.

Each renderer can define its own configuration options which are loaded by the plastex script. This is done in a file named `Config.py` in the renderer directory. This file must define a variable named *config* which is a ConfigManager instance, as described in Section 6.2. Inspiration can be drawn from the file defining the global configuration which is `plasTeX/Config.py`.

For instance, one could add a file `plasTeX/Renderers/SimpleRenderer/Config.py` containing:

```
from plasTeX.ConfigManager import *

config = ConfigManager()

section = config.add_section('simplerenderer')

config.add_category('simplerenderer', 'Simple Renderer Options')

section['my-option'] = StringOption(
    """ My option """,
    options='--my-option',
    category='simplerenderer',
    default='',
)
```

Options values are attached to the document currently rendered. For instance, in the *default* method implemented in Section 5.1, which takes a node argument, one could access the value of the option defined above as `node.ownerDocument.config['simplerenderer']['my-option']`.

5.2 Renderable Objects

The `Renderable` class is the real workhorse of the rendering process. It traverses the document object, looks up the appropriate rendering methods in the renderer, and generates the output files. It also invokes the image generating process when needed for parts of a document that cannot be rendered in the given output format.

Most of the work of the `Renderable` class is done in the `__unicode__` method. This is rather convenient since each of the rendering methods in the renderer are required to return a unicode object. When the `unicode` function is called with a renderable object as its argument, the document traversal begins for that node. This traversal includes iterating through each of the node's child nodes, and looking up and calling the appropriate rendering method in the renderer. If the child node is configured to generate a new output file, the file is created and the rendered output is written to it; otherwise, the rendered output is appended to the rendered output of previous nodes. Once all of the child nodes have been rendered, the unicode object containing that output is returned. This recursive process continues until the entire document has been rendered.

There are a few useful things to know about renderable objects such as how they determine which rendering method to use, when to generate new files, what the filenames will be, and how to generate images. These things are discussed below.

5.2.1 Determining the Correct Rendering Method

Looking up the correct rendering method is quite straight-forward. If the node is a text node, the `textDefault` attribute on the renderer is used. If it is not a text node, then the node's name determines the key name in the renderer. In most cases, the node's name is the same name as the LaTeX macro that created it. If the macro used some type of modifier argument (i.e. *, +, -), a name with that modifier applied to it is also searched for first. For example, if you used the `tablular*` environment in your LaTeX document, the renderer will look for "tabular*" first, then "tabular". This allows you to use different rendering methods for modified and unmodified macros. If no rendering method is found, the method in the renderer's `default` attribute is used.

5.2.2 Generating Files

Any node in a document has the ability to generate a new file. During document traversal, each node is queried for a filename. If a non-*None* is returned, a new file is created for the content of that node using the given filename. The querying for the filename is simply done by accessing the `filename` property of the node. This property is added to the node's namespace during the mixin process. The default behavior for this property is to only return filenames for sections with a level less than the split-level given in the configuration (see section 2.1.5). The filenames generated by this routine are very flexible. They can be statically given names, or names based on the ID and/or title, or simply generically numbered. For more information on configuring filenames see section 2.1.5.

While the filenaming mechanism is very powerful, you may want to give your files names based on some other information. This is possible through the `filenameoverride` attribute. If the `filenameoverride` is set, the name returned by that attribute is used as the filename. The string in `filenameoverride` is still processed in the same way as the filename specifier in the configuration so that you can use things like the ID or title of the section in the overridden filename.

The string used to specify filenames can also contain directory paths. This is not terribly useful at the moment since there is no way to get the relative URLs between two nodes for linking purposes.

If you want to use a filename override, but want to do it conditionally you can use a Python property to do this. Just calculate the filename however you wish, if you decide that you don't want to use that filename then raise an `AttributeError` exception. An example of this is shown below.

```
class mymacro{Command):
    args = '[ filename:str ] self'
    @property
    def filenameoverride(self):
        # See if the attributes dictionary has a filename
        if self.attributes['filename'] is not None:
            return self.attributes['filename']
        raise AttributeError, 'filenameoverride'
```

Note: The filename in the `filenameoverride` attribute must contain any directory paths as well as a file extension.

5.2.3 Generating Images

Not all output types that you might render are going to support everything that LaTeX is capable of. For example, HTML has no way of representing equations, and most output types won't be capable of rendering LaTeX's `picture` environment. In cases like these, you can let plasTeX generate images of the document node. Generating images is done with a subclass of `plasTeX.Imagers.Imager`. The imager is responsible for creating a LaTeX document from the requested document fragments, compiling the document and converting each page of the output document into individual images. Currently, there are two `Imager` subclasses included with plasTeX. Each of them use the standard LaTeX compiler to generate a DVI file. The DVI file is then converted into images using one of the available imagers (see section 2.1.6 on how to select different imagers).

To generate an image of a document node, simply access the `image` property during the rendering process. This property will return an `plasTeX.Imagers.Image` instance. In most cases, the image file will not be available until the rendering process is finished since most renderers will need the generated LaTeX document to be complete before compiling it and generating the final images.

The example below demonstrates how to generate an image for the `equation` environment.

```
# Import renderer from first renderer example
from MyRenderer import Renderer

from plasTeX.TeX import TeX

def handle_equation(node):
    return u'<div><img src="%s"/></div>' % node.image.url

# Instantiate a TeX processor and parse the input text
tex = TeX()
tex.input(r'''
\documentclass{book}
\begin{document}

Previous paragraph.

\begin{equation}
\Sigma_{x=0}^{x+n} = \beta^2
\end{equation}

Next paragraph.

\end{document}
''')
document = tex.parse()

# Instantiate the renderer
renderer = Renderer()

# Insert the rendering method into all of the environments that might need it
renderer['equation'] = handle_equation
renderer['displaymath'] = handle_equation
renderer['eqnarray'] = handle_equation

# Render the document
renderer.render(document)
```

The rendered output looks like the following, and the image is generated is located in 'images/img-0001.png'.

```
<document>
<par>
Previous paragraph.
</par><par>
<div><img src="images/img-0001.png"/></div>
</par><par>
Next paragraph.
</par>
</document>
```

The names of the image files are determined by the document's configuration. The filename generator is very powerful, and is in fact, the same filename generator used to create the other output filenames. For more information on customizing the image filenames see section 2.1.6.

In addition, the image types are customizable as well. plasTEX uses the Python Imaging Library (PIL) to do the final cropping and saving of the image files, so any image format that PIL supports can be used. The format that PIL saves the images in is determined by the file extension in the generated filenames, so you must use a file extension that PIL recognizes.

It is possible to write your own `Imager` subclass if necessary. See the `Imager` API documentation for more information (see 6.7).

5.2.4 Generating Vector Images

If you have a vector imager configured (such as dvisvg or dvisvgm), you can generate a vector version of the requested image as well as a bitmap. The nice thing about vector versions of images is that they can scale infinitely and not loose resolution. The bad thing about them is that they are not as well supported in the real world as bitmaps.

Generating a vector image is just as easy as generating a bitmap image, you simply access the `vectorImage` property of the node that you want an image of. This will return an `plasTeX.Imagers.Image` instance that corresponds to the vector image. A bitmap version of the same image can be accessed through the `image` property of the document node or the `bitmap` variable of the vector image object.

Everything that was described about generating images in the previous section is also true of vector images with the exception of cropping. plasTEX does not attempt to crop vector images. The program that converts the LATEX output to a vector image is expected to crop the image down to the image content. plasTEX uses the information from the bitmap version of the image to determine the proper depth of the vector image.

5.2.5 Static Images

There are some images in a document that don't need to be generated, they simply need to be copied to the output directory and possibly converted to an appropriate formate. This is accomplished with the `imageoverride` attribute. When the `image` property is accessed, the `imageoverride` attribute is checked to see if an image is already available for that node. If there is, the image is copied to the image output directory using a name generated using the same method as described in the previous section. The image is copied to that new filename and converted to the appropriate image format if needed. While it would be possible to simply copy the image over using the same filename, this may cause filename collisions depending on the directory structure that the original images were store in.

Below is an example of using `imageoverride` for copying stock icons that are used throughout the document.

```
from plasTeX import Command

class dangericon(Command):
    imageoverride = 'danger.gif'

class warningicon(Command):
    imageoverride = 'warning.gif'
```

It is also possible to make `imageoverride` a property so that the image override can done conditionally. In the case where no override is desired in a property implementation, simply raise an `AttributeError` exception.

5.3 Page Template Renderer

The Page Template (PT) renderer is a renderer for plasTEX document objects that supports various page template engines such as Zope Page Templates (ZPT), Jinja2 templates, Cheetah templates, Kid templates, Genshi templates, Python string templates, as well as plain old Python string formatting. It is also possible to add support for other template engines. Note that all template engines except ZPT, Python formats, and Python string templates must be installed in your Python installation. They are not included. In particular the Jinja2 template engine must be installed in order to use the HTML5 renderer.

The ZPT engine is used for all of the plasTEX delivered templates in the XHTML renderer; however, the other templates work in a very similar way. The actual ZPT implementation used is SimpleTAL (`http://www.owlfish.com/software/simpleTAL/`). This implementation implements almost all of the ZPT API and is very stable. However, some changes were made to this package to make it more convenient to use within plasTEX. These changes are discussed in detail in the ZPT Tutorial (see section 5.3.3).

Since the above template engines can be used to generate any form of XML or HTML, the PT renderer is a general solution for rendering XML or HTML from a plasTEX document object. When switching from one DTD to another, you simply need to use a different set of templates.

As in all `Renderer`-based renderers, each key in the PT renderer returns a function. These functions are actually generated when the template files are parsed by the PT renderer. As is the case with all rendering methods, the only argument is the node to be rendered, and the output is a unicode object containing the rendered output. In addition to the rendering methods, the `textDefault` method escapes all characters that are special in XML and HTML (i.e. <, >, and &).

The following sections describe how templates are loaded into the renderer, how to extend the set of templates with your own, as well as a theming mechanism that allows you to apply different looks to output types that are visual (e.g. HTML).

5.3.1 Defining and Using Templates

Note: If you are not familiar with the ZPT language, you should read the tutorial in section 5.3.3 before continuing in this section. See the links in the previous section for documentation on the other template engines.

By default, templates are loaded from the directory where the renderer module was imported from. In addition, the templates from each of the parent renderer class modules are also loaded. This makes it very easy to extend a renderer and add just a few new templates to support the additions that were made.

The template files in the module directories can have three different forms. The first is HTML. HTML templates must have an extension of '.htm' or '.html'. These templates are compiled using SimpleTAL's HTML compiler. XML templates, the second form of template, uses SimpleTAL's XML compiler, so they must be well-formed XML fragments. XML templates must have the file extension '.xml', '.xhtml', or '.xhtm'. In any case, the basename of the template file is used as the key to store the template in the renderer. Keep in mind that the names of the keys in the renderer correspond to the node names in the document object.

The extensions used for all templating engines are shown in the table below.

Engine	**Extension**	**Output Type**
ZPT	.html, .htm, .zpt	HTML
	.xhtml, .xhtm, .xml	XML/XHTML
Jinja2	.jinja2	Any
Python string formatting	.pyt	Any
Python string templates	.st	Any
Kid	.kid	XML/XHTML
Cheetah	.che	XML/XHTML
Genshi	.gen	HTML

The file listing below is an example of a directory of template files. In this case the templates correspond to nodes in the document created by the `description` environment, the `tabular` environment, `\textbf`, and `\textit`.

```
description.xml
tabular.xml
textbf.html
textit.html
```

Since there are a lot of templates that are merely one line, it would be inconvenient to have to create a new file for each template. In cases like this, you can use the '.zpts' extension for collections of ZPT templates, or the '.jinja2s' extension for collections of Jinja2 templates, or more generally '.pts' for collections of various template types. Files with this extension have multiple templates in them. Each template is separated from the next by the template metadata which includes things like the name of the template, the type (xml, html, or text), and can also alias template names to another template in the renderer. The following metadata names are currently supported.

Name	Purpose
engine	the name of the templating engine to use. At the time of this writing, the value could be zpt, tal (same as zpt), html (ZPT HTML template), xml (ZPT XML template), jinja2, python (Python formatted string), string (Python string template), kid, cheetah, or genshi.
name	the name or names of the template that is to follow. This name is used as the key in the renderer, and also corresponds to the node name that will be rendered by the template. If more than one name is desired, they are simply separated by spaces.
type	the type of the template: xml, html, or text. XML templates must contain a well-formed XML fragment. HTML templates are more forgiving, but do not support all features of ZPT (see the SimpleTAL documentation).
alias	specifies the name of another template that the given names should be aliased to. This allows you to simply reference another template to use rather than redefining one. For example, you might create a new section heading called `\introduction` that should render the same way as `\section`. In this case, you would set the name to "introduction" and the alias to "section".

There are also some defaults that you can set at the top of the file that get applied to the entire file unles overridden by the meta-data on a particular template.

Name	Purpose
default-engine	the name of the engine to use for all templates in the file.
default-type	the default template type for all templates in the file.

The code sample below shows the basic format of a zpts file.

```
name: textbf bfseries
<b tal:content="self">bold content</b>

name: textit
<i tal:content="self">italic content</i>

name: introduction introduction*
alias: section

name: description
type: xml
<dl>
<metal:block tal:repeat="item self">
    <dt tal:content="item/attributes/term">definition term</dt>
    <dd tal:content="item">definition content</dd>
</metal:block>
</dl>
```

The code above is a zpts file that contains four templates. Each template begins when a line starts with "name:". Other directives have the same format (i.e. the name of the directive followed by a colon) and must immediately follow the name directive. The first template definition actually applies to two types of nodes *textbf* and *bfseries*. You can specify ony number of names on the name line. The third template isn't a template at all; it is an alias. When an alias is specified, the name (or names) given use the same template as the one specified in the alias directive. Notice also that starred versions of a macro can be specified separately. This means that they can use a different template than the un-starred versions of the command. The last template is just a simple XML formatted template. By default, templates in a zpts file use the HTML compiler in SimpleTAL. You can specify that a template is an XML template by using the type directive.

Here is an example of using various templates engines in a single file.

```
name: equation
engine: jinja2
<div class="equation" id="{{ obj.id }}">
  <span class="equation_label">{{ obj.ref }}</span>
  {{ obj }}
</div>

name: textbf
engine: python
<b>%(self)s</b>

name: textit
engine: string
<i>${self}</i>

name: textsc
engine: cheetah
<span class="textsc">${here}</span>

name: textrm
engine: kid
<span class="textrm" py:content="XML(unicode(here))">normal text</span>

name: textup
engine: genshi
<span class="textup" py:content="markup(here)">upcase text</span>
```

There are several variables inserted into the template namespace. Here is a list of the variables and the templates that support them.

Object	ZPT/Python Formats/String Template	Jinja2	Cheetah	Kid/Genshi
document node	*self* or *here*	*obj* or *here*	*here*	*here*
parent node	*container*	*container*	*container*	*container*
document config	*config*	*config*	*config*	*config*
template instance	*template*			
renderer instance	*templates*	*templates*	*templates*	*templates*

You'll notice that Kid and Genshi templates require some extra processing of the variables in order to get the proper markup. By default, these templates escape characters like <, >, and &. In order to get HTML/XML markup from the variables you must wrap them in the code shown in the example above. Hopefully, this limitation will be removed in the future.

When using Jinja2 templates, the default configuration trims white spaces before and after template tags (see trim_blocks and lstrip_blocks in Jinja2's documentation). Also, when developing Jinja2 templates, inserting `{{ debug() }}` will launch a python debugger session to allow inspection of the *context* variable during rendering.

Template Overrides

It is possible to override the templates located in a renderer's directory with templates defined elsewhere. This is done using the `*TEMPLATES` environment variable. The "*" in the name `*TEMPLATES` is a wildcard and must be replaced by the name of the renderer. For example, if you are using the XHTML renderer, the environment variable would be `XHTMLTEMPLATES`. For the PageTemplate renderer, the environment variable would be `PAGETEMPLATETEMPLATES`.

The format of this variable is the same as that of the PATH environment variable which means that you can put multiple directory names in this variable. In addition, the environment variables for each of the parent renderers is also used, so that you can use multiple layers of template directories.

You can actually create an entire renderer just using overrides and the PT renderer. Since the PT renderer doesn't actually define any templates, it is just a framework for defining other XML/HTML renderers, you can simply load the PT renderer and set the PAGETEMPLATETEMPLATES environment variable to the locations of your templates. This method of creating renderers will work for any XML/HTML that doesn't require any special post-processing.

5.3.2 Defining and Using Themes

In addition to the templates that define how each node should be rendered, there are also templates that define page layouts. Page layouts are used whenever a node in the document generates a new file. Page layouts generally include all of the markup required to make a complete document of the desired DTD, and may include things like navigation buttons, tables of contents, breadcrumb trails, etc. to link the current file to other files in the document.

When rendering files, the content of the node is generated first, then that content is wrapped in a page layout. The page layouts are defined the same way as regular templates; however, they all include "-layout" at the end of the template name. For example the sectioning commands in LaTeX would use the layout templates "section-layout", "subsection-layout", "subsubsection-layout", etc. Again, these templates can exist in files by themselves or multiply specified in a zpts file. If no layout template exists for a particular node, the template name "default-layout" is used.

Since there can be several themes defined within a renderer, theme files are stored in a subdirectory of a renderer directory. This directory is named 'Themes'. The 'Themes' directory itself only contains directories that correspond to the themes themselves where the name of the directory corresponds to the name of the theme. These theme directories generally only consist of the layout files described above, but can override other templates as well. Below is a file listing demonstrating the structure of a renderer with multiple themese.

```
# Renderer directory: contains template files
XHTML/

# Theme directory: contains theme directories
XHTML/Themes/

# Theme directories: contain page layout templates
XHTML/Themes/default/
XHTML/Themes/fancy/
XHTML/Themes/plain/
```

Note: If no theme is specified in the document configuration, a theme with the name "default" is used.

Since all template directories are created equally, you can also define themes in template directories specified by environment variables as described in section 5.3.1. Also, theme files are searched in the same way as regular templates, so any theme defined in a renderer superclass' directory is valid as well.

5.3.3 Zope Page Template Tutorial

The Zope Page Template (ZPT) language is actually just a set of XML attributes that can be applied to markup of an DTD. These attributes tell the ZPT interpreter how to process the element. There are seven different attributes that you can use to direct the processing of an XML or HTML file (in order of evaluation): define, condition, repeat, content, replace, attributes, and omit-tag. These attributes are described in section 5.3.3. For a more complete description, see the official ZPT documentation at http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/ZPT.stx.

Template Attribute Language Expression Syntax (TALES)

The Template Attribute Language Expression Syntax (TALES) is used by the attribute language described in the next section. The TALES syntax is used to evaluate expressions based on objects in the template namespace. The results of these expressions can be used to define variables, produce output, or be used as booleans. There are also several operators used to modify the behavior or interpretation of an expression. The expressions and their modifiers are described below.

path: operator A "path" is the most basic form on an expression in ZPT. The basic form is shown below.

```
[path:]string [ | TALES expression ]
```

The *path:* operator is actually optional on all paths. Leaving it off makes no difference. The "string" in the above syntax is a '/' delimited string of names. Each name refers to a property of the previous name in the string. Properties can include attributes, methods, or keys in a dictionary. These properties can in turn have properties of their own. Some examples of paths are shown below.

```
# Access the parentNode attribute of chapter, then get its title
chapter/parentNode/title

# Get the key named 'foo' from the dictionary bar
bar/foo

# Call the title method on the string in the variable booktitle
booktitle/title
```

It is possible to specify multiple paths separated by a pipe (|). These paths are evaluated from left to right. The first one to return a non-None value is used.

```
# Look for the title on the current chapter node as well as its parents
chapter/title | chapter/parentNode/title | chapter/parentNode/parentNode/title

# Look for the value of the option otherwise get its default value
myoptions/encoding | myoptions/defaultencoding
```

There are a few keywords that can be used in place of a path in a TALES expression as well.

Name	Purpose
nothing	same as *None* in Python
default	keeps whatever the existing value of the element or attribute is
options	dictionary of values passed in to the template when instatiated
repeat	the repeat variable (see 5.3.3)
attrs	dictonary of the original attributes of the element
CONTEXTS	dictionary containing all of the above

exists: operator This operator returns true if the path exists. If the path does not exist, the operator returns false. The syntax is as follows.

```
exists:path
```

The "path" in the code above is a path as described in section 5.3.3. This operator is commonly combined with the not: operator.

nocall: operator By default, if a property that is retrieved is callable, it will be called automatically. Using the nocall: operator, prevents this execution from happening. The syntax is shown below.

```
nocall:path
```

not: operator The not: operator simply negates the boolean result of the path. If the path is a boolean true, the not: operator will return false, and vice versa. The syntax is shown below.

```
not:path
```

string: operator The string: operator allows you to combine literal strings and paths into one string. Paths are inserted into the literal string using a syntax much like that of Python Templates: $path or ${path}. The general syntax is:

```
string:text
```

Here are some examples of using the string: operator.

```
string:Next - ${section/links/next}
string:($pagenumber)
string:[${figure/number}] ${figure/caption}
```

python: operator The python: operator allows you to evaluate a Python expression. The syntax is as follows.

```
python:python-code
```

The "python-code" in the expression above can include any of the Python built-in functions and operators as well as four new functions that correspond to the TALES operators: path, string, exists, and nocall. Each of these functions takes a string containing the path to be evaluated (e.g. path('foo/bar'), exists('chapter/title'), etc.).

When using Python operators, you must escape any characters that would not be legal in an XML/HTML document (i.e. <>&). For example, to write an expression to test if a number was less than or greater than two numbers, you would need to do something like the following example.

```
# See if the figure number is less than 2 or greater than 4
python: path('figure/number') &lt; 2 or path('figure/number') &gt; 4
```

stripped: operator The stripped: operator only exists in the SimpleTAL distribution provided by plasTEX. It evaluates the given path and removes any markup from that path. Essentially, it is a way to get a plain text representation of the path. The syntax is as follows.

```
stripped:path
```

Template Attribute Language (TAL) Attributes

tal:define The `tal:define` attribute allows you to define a variable for use later in the template. Variables can be specifies as local (only for use in the scope of the current element) or global (for use anywhere in the template). The syntax of the define attribute is shown below.

```
tal:define="[ local | global ] name expression [; define-expression ]"
```

The define attributes sets the value of "name" to "expression." By default, the scope of the variable is local, but can be specified as global by including the "global" keyword before the name of the variable. As shown in the grammar above, you can specify multiple variables in one `tal:define` attribute by separating the define expressions by semi-colons.

Examples of using the `tal:define` attribute are shown belaw.

```
<p tal:define="global title document/title;
               next self/links/next;
               previous self/links/previous;
               length python:len(self);
               up string:Up - ${self/links/up}">
...
</p>
```

tal:condition The `tal:condition` attribute allows you to conditionally include an element. The syntax is shown below.

```
tal:condition="expression"
```

The `tal:condition` attribute is very simple. If the expression evaluates to true, the element and its children will be evaluated and included in the output. If the expression evaluates to false, the element and its children will not be evaluated or included in the output. Valid expressions for the `tal:condition` attribute are the same as those for the expressions in the `tal:define` attribute.

```
<p tal:condition="python:len(self)">
    <b tal:condition="self/caption">Caption for paragraph</b>
    ...
</p>
```

tal:repeat The `tal:repeat` attribute allows you to repeat an element multiple times; the syntax is shown below.

```
tal:repeat="name expression"
```

When the `tal:repeat` attribute is used on an element, the result of"expression" is iterated over, and a new element is generated for each item in the iteration. The value of the current item is set to "name" much like in the `tal:define` attribute.

Within the scope of the repeated element, another variable is available: *repeat*. This variable contains several properties related to the loop.

Name	Purpose
index	number of the current iteration starting from zero
number	number of the current iteration starting from one
even	is true if the iteration number is even
odd	is true if the iteration number is odd
start	is true if this is the first iteration
end	is true if this is the last iteration; This is never true if the repeat expression returns an iterator
length	the length of the sequence being iterated over; This is set to *sys.maxint* for iterators.
letter	lower case letter corresponding to the current iteration number starting with 'a'
Letter	upper case letter corresponding to the current iteration number starting with 'A'
roman	lower case Roman numeral corresponding to the current iteration number starting with 'i'
Roman	upper case Roman numeral corresponding to the current iteration number starting with 'I'

To access the properties listed above, you must use the property of the *repeat* variable that corresponds to the repeat variable name. For example, if your repeat variable name is "item", you would access the above variables using the expressions *repeat/item/index*, *repeat/item/number*, *repeat/item/even*, etc.

A simple example of the `tal:repeat` attribute is shown below.

```
<ol>
<li tal:repeat="option options" tal:content="option/name">option name</li>
</ol>
```

One commonly used feature of rendering tables is alternating row colors. This is a little bit tricky with ZPT since the `tal:condition` attribute is evaluated before the `tal:repeat` directive. You can get around this by using the `metal:` namespace. This is the namespace used by ZPT's macro language[3] You can create another element around the element you want to be conditional. This wrapper element is simply there to do the iterating, but is not included in the output. The example below shows how to do alternating row colors in an HTML table.

[3]The macro language isn't discussed here. See the official ZPT documentation for more information.

```
<table>
<metal:block tal:repeat="employee employees">
<!-- even rows -->
<tr tal:condition="repeat/employee/even" style="background-color: white">
    <td tal:content="employee/name"></td>
    <td tal:content="employee/title"></td>
</tr>
<!-- odd rows -->
<tr tal:condition="repeat/employee/odd" style="background-color: gray">
    <td tal:content="employee/name"></td>
    <td tal:content="employee/title"></td>
</tr>
</metal:block>
</table>
```

tal:content The `tal:content` attribute evaluates an expression and replaces the content of the element with the result of the expression. The syntax is shown below.

```
tal:content="[ text | structure ] expression"
```

The *text* and *structure* options in the `tal:content` attribute indicate whether or not the content returned by the expression should be escaped (i.e. "&<> replaced by ", &, <, and >, respectively). When the *text* option is used, these special characters are escaped; this is the default behavior. When the *structure* option is specified, the result of the expression is assumed to be valid markup and is not escaped.

In SimpleTAL, the default behavior is the same as using the *text* option. However, in plasTEX, 99.9% of the time the content returned by the expression is valid markup, so the default was changed to *structure* in the SimpleTAL package distributed with plasTEX.

tal:replace The `tal:replace` attribute is much like the `tal:content` attribute. They both evaluate an expression and include the content of that expression in the output, and they both have a *text* and *structure* option to indicate escaping of special characters. The difference is that when the `tal:replace` attribute is used, the element with the `tal:replace` attribute on it is not included in the output. Only the content of the evaluated expression is returned. The syntax of the `tal:replace` attribute is shown below.

```
tal:replace="[ text | structure ] expression"
```

tal:attributes The `tal:attributes` attribute allows you to programatically create attributes on the element. The syntax is shown below.

```
tal:attributes="name expression [; attribute-expression ]"
```

The syntax of the `tal:attributes` attribute is very similar to that of the `tal:define` attribute. However, in the case of the `tal:attributes` attribute, the name is the name of the attribute to be created on the element and the expression is evaluated to get the value of the attribute. If an error occurs or *None* is returned by the expression, then the attribute is removed from the element.

Just as in the case of the `tal:define` attribute, you can specify multiple attributes separated by semi-colons (;). If a semi-colon character is needed in the expression, then it must be represented by a double semi-colon (;;).

An example of using the `tal:attributes` is shown below.

```
<a tal:attributes="href self/links/next/url;
                   title self/links/next/title">link text</a>
```

tal:omit-tag The `tal:omit-tag` attribute allows you to conditionally omit an element. The syntax is shown below.

```
tal:omit-tag="expression"
```

If the value of "expression" evaluates to true (or is empty), the element is omitted; however, the content of the element is still sent to the output. If the expression evaluates to false, the element is included in the output.

5.4 XHTML Renderer

The XHTML renderer is a subclass of the Page Template Renderer (section 5.3). Since the Page Template Renderer can render any variant of XML or HTML, the XHTML renderer has very little to do in the Python code. Almost all of the additional processing in the XHTML renderer has to do with generated images. Since HTML cannot render LaTeX's vector graphics or equations natively, they are converted to images. In order for inline equations to line up correctly with the text around them, CSS attributes are used to adjust the vertical alignment. Since the images aren't generated until after all of the document has been rendered, this CSS information is added in post-processing (i.e. the `cleanup` method).

In addition to the processing of images, all characters with a ordinal greater than 127 are converted into numerical entities. This should prevent any rendering problems due to unknown encodings.

Most of the work in this renderer was in creating the templates for every LaTeX construct. Since this renderer was intended to be the basis of all HTML-based renderers, it must be capable of rendering all LaTeX constructs; therefore, there are ZPT templates for every LaTeX command, and the commands in some common LaTeX packages.

While the XHTML renderer is fairly complete when it comes to standard LaTeX, there are many packages which are not currently supported. To add support for these packages, templates (and possibly Python based macros; section 4) must be created.

5.4.1 Themes

The theming support in the XHTML renderer is the same as that of the Page Template Renderer. Any template directory can have a subdirectory called 'Themes' which contains theme directories with sets of templates in them. The names of the directories in the 'Themes' directory corresponds to the name of the theme. There are currently two themes included with plasTeX: default and plain. The default theme is a minor variation of the one used in the Python 1.6 documentation. The plain theme is a theme with no extra navigation bars.

5.5 HTML5 Renderer

The HTML5 Renderer is a subclass of the Page Template Renderer (Section 5.3). Therefore most of the work is done in its collection of templates, all written using the Jinja2 template engine. In particular Jinja2 must be installed on your system to use this renderer.

In addition, this renderer allows packages to override certain templates, add css or javascript files, and push output files through various filters, see Section 5.5.1.

Options described in Section 2.1.7 provide easy ways to customize output. In particular the **--extra-css** option allows to override CSS styles. Since the generated HTML contains no inline style (except for some size specification for images and tables), these CSS overrides allow to completely change the output style.

For large scale CSS changes, one can create a new CSS theme, either from scratch or by customizing an existing theme. Existing themes are generated using the CSS extension language SASS (see `http://sass-lang.com/`). Their sources are located in 'plasTeX/Renderers/HTML5/sources/sass/'. One way to use customize them is to do the following (assuming SASS is available on your system). Copy the above directory somewhere else, say in 'mysass', copy 'theme-blue.scss' to 'theme-custom.scss' and replace "blue" in the first line by "custom", copy '_variables_blue.scss' to '_variables_custom.scss', modify a number of values in this file and compile using

```
sass --update --sourcemap=none mysass:build
```

This will create 'build/theme-custom.css' which can be copied to your project and used by plasTeX using

```
plastex ---no-theme-css --extra-css=theme-custom.css mytexfile.tex
```

Of course one can also change other 'scss' files for larger changes. Note that distributed theme css also went through autoprefixer (`https://autoprefixer.github.io/`) to ensure cross browser compatibility and cssnano (`http://cssnano.co/`) to reduce their size (all those steps are performed by 'plasTeX/Renderers/HTML5/sources/build-css.sh'). If CSS modifications are not enough, one can override templates as discussed in Section 5.3.1.

The normal way to handle mathematics in this renderer is to use MathJax (see `https://www.mathjax.org`) to render mathematics on client side. This is controlled by the **--use-mathjax** option which is set to true by default. Option **--mathjax-url** indicates where to find the MathJax library. By default it uses a CDN which ensures using the latest version but prevents offline use. Instead of client-side rendering, one can use filters to handle mathematics on the author side. For instance one can use mathjax-node (`https://github.com/mathjax/MathJax-node`) or KaTeX (`https://khan.github.io/KaTeX/`). In this case, one can disable inclusion of MathJax using option **--no-mathjax** and use option **--filters** to call the author-side mathematics renderer.

5.5.1 Interactions with packages

This section is useful for packages writers. The HTML5 Renderer allows packages to interact with the rendering process in several ways.

Extra input files

Inside the main renderer directory 'plasTeX/Renderers/HTML5/' there are directories 'pkgTemplates', 'pkgCss' and 'pkgJs' where any package can add a subdirectory. These must be registered in the package `ProcessOptions` function by adding the subdirectory name in *document.userdata['pkgTemplates']*, *document.userdata['pkgCss']* or *document.userdata['pkgJs']*. Directories inside 'pkgTemplates' contain templates which will override regular renderer templates. Directories inside 'pkgCss' contain CSS files which will be loaded (at least when using the default layout),

in alphabetical order, after the theme CSS files, but before user defined extra CSS files. Directories inside 'pkgJs' contain Javascript files which will be loaded (at least when using the default layout), in alphabetical order, after the theme Javascript files, but before user defined extra Javascript files. For instance, if a package 'mypackage.py' wants to use all three possibilities, it must create, inside 'plasTeX/Renderers/HTML5/' subdirectories 'pkgTemplates/mypackage', 'pkgCss/mypackage', 'pkgJs/mypackage' and contain a `ProcessOptions` function including the following:

```
def ProcessOptions(options, document):
    document.userdata.setdefault('pkgTemplates', []).append('mypackage')
    document.userdata.setdefault('pkgCss', []).append('mypackage')
    document.userdata.setdefault('pkgJs', []).append('mypackage')
```

In the example above, notice how `setdefault` is used to ensure creation of dictionary keys if needed (this happens to the first loaded package which wants to use these features) but without overwriting what could have been registered by earlier packages.

Extra output files and filters

The HTML5 Renderer allows packages to produce extra output files and define filters that are applied to any output file.

In order to create new output files, any package can register a callback function taking a document as input a returning a list of created file names by adding it to the current document *userdata['preCleanupCallbacks']* list. Initially this key does not exist in *userdata* hence each package that uses this possibility must use the `setdefault` method, as explained in the previous section. As an example, let us write a package which adds some document statistics to an extra output file 'stats.html' (we will not try to produce valid html below, only illustrate our point). The package module only needs to contain the following.

```
import os

def ProcessOptions(options, document):
    def makeStats(document):
        nbChap = len(document.getElementsByTagName('chapter'))
        nbSec = len(document.getElementsByTagName('section'))
        with open('stats.html', 'w') as file:
            file.write("%d chapters and %d sections" % (nbChap, nbSec))
        return ['stats.html']
    document.userdata.setdefault('preCleanupCallbacks', []).append(makeStats)
```

As suggested by the key name, those pre-cleanup callbacks are called before the renderer `cleanup` method. Packages can also register filters to be applied during the cleanup process. Those filters are functions that take a document and a string to filter and return the filtered string (using the document object to provide context if needed). They will be called on the content of each rendered file, after all files have been rendered. Packages can register such filter in *document.userdata['processFileContents']*. Again this key does not exist if no package wants to use it, so it should be used using `setdefault` as in the above example. This method can be used to call an external renderer to handle some complex environment (e.g. a TikZ picture).

5.6 tBook Renderer

Not yet implemented.

5.7 DocBook Renderer

Not yet implemented.

CHAPTER

SIX

PLASTEX Frameworks and APIs

6.1 plasTeX — The Python Macro and Document Interfaces

While plasTEX does a respectable job expanding LATEX macros, some macros may be too complicated for it to handle. These macros may have to be re-coded as Python objects. Another reason you may want to use Python-based macros is for performance reasons. In most cases, macros coded using Python will be faster than those expanded as true LATEX macros.

The API for Python macros is much higher-level than that of LATEX macros. This has good and bad ramifications. The good is that most common forms of LATEX macros can be parsed and processed very easily using Python code which is easier to read than LATEX code. The bad news is that if you are doing something that isn't common, you will have more work to do. Below is a basic example.

```
from plasTeX import Command

class mycommand(Command):
    """ \mycommand[name]{title} """
    args = '[ name ] title'
```

The code above demonstrates how to create a Python-based macro corresponding to LATEX macro with the form `\mycommand[name]{title}` where 'name' is an optional argument and 'title' is a mandatory argument. In the Python version of the macro, you simply declare the arguments in the `args` attribute as they would be used in the LATEX macro, while leaving the braces off of the mandatory arguments. When parsed in a LATEX document, an instance of the class `mycommand` in created and the arguments corresponding to 'name' and 'title' are set in the `attributes` dictionary for that instance. This is very similar to the way an XML DOM works, and there are more DOM similarities yet to come. In addition, there are ways to handle casting of the arguments to various data types in Python. The API documentation below goes into more detail on these and many more aspects of the Python macro API.

6.1.1 Macro Objects

class Macro()

The `Macro` class is the base class for all Python based macros although you will generally want to subclass from `Command` or `Environment` in real-world use. There are various attributes and methods that affect how Python macros are parsed, constructed and inserted into the resulting DOM. These are described below.

args

specifies the arguments to the LATEX macro and their data types. The `args` attribute gives you a very simple, yet extremely powerful way of parsing LATEX macro arguments and converting them into Python objects. Once parsed, each LATEX macro argument is set in the `attributes` dictionary of the Python instance using the name given in the `args` string. For example, the following `args` string will direct plasTEX to parse two mandatory

arguments, ‘id’ and ‘title’, and put them into the `attributes` dictonary.

```
args = 'id title'
```

You can also parse optional arguments, usually surrounded by square brackets ([ ]). However, in plasTEX, any arguments specified in the `args` string that aren't mandatory (i.e. no braces surrounding it) are automatically considered optional. This may not truly be the case, but it doesn't make much difference. If they truly are mandatory, then your LATEX source file will always have them and plasTEX will simply always find them even though it considers them to be optional.

Optional arguments in the `args` string are surround by matching square brackets ([ ]), angle brackets (< >), or parentheses (( )). The name for the attribute is placed between the matching symbols as follows:

```
args = '[ toc ] title'
args = '( position ) object'
args = '< markup > ref'
```

You can have as many optional arguments as you wish. It is also possible to have optional arguments using braces ({ }), but this requires you to change TEX's category codes and is not common.

Modifiers such as asterisks (*) are also allowed in the `args` string. You can also use the plus (+) and minus (-) signs as modifiers although these are not common. Using modifiers can affect the incrementing of counters (see the `parse()` method for more information).

In addition to specifying which arguments to parse, you can also specify what the data type should be. By default, all arguments are processed and stored as document fragments. However, some arguments may be simpler than that. They may contain an integer, a string, an ID, etc. Others may be collections like a list or dictionary. There are even more esoteric types for mostly internal use that allow you to get unexpanded tokens, TEX dimensions, and the like. Regardless, all of these directives are specified in the same way, using the typecast operator: ‘:’. To cast an argument, simply place a colon (:) and the name of the argument type immediately after the name of the argument. The following example casts the ‘filename’ argument to a string.

```
args = 'filename:str'
```

Parsing compound arguments such as lists and dictionaries is very similar.

```
args = 'filenames:list'
```

By default, compound arguments are assumed to be comma separated. If you are using a different separator, it is specified in parentheses after the type.

```
args = 'filenames:list(;)'
```

Again, each element element in the list, by default, is a document fragment. However, you can also give the data type of the elements with another typecast.

```
args = 'filenames:list(;):str'
```

Parsing dictionaries is a bit more restrictive. plasTEX assumes that dictionary arguments are always key-value pairs, that the key is always a string and the separator between the key and value is an equals sign (=). Other than that, they operate in the same manner.

A full list of the supported data types as well as more examples are discussed in section 4.

`argSource`
the source for the LATEX arguments to this macro. This is a read-only attribute.

arguments
gives the arguments in the `args` attribute in object form (i.e. `Argument` objects). **Note:** This is a read-only attribute. **Note:** This is generally an internal-use-only attribute.

blockType
indicates whether the macro node should be considered a block-level element. If true, this node will be put into its own paragraph node (which also has the `blockType` set to *True*) to make it easier to generate output that requires block-level to exist outside of paragraphs.

counter
specifies the name of the counter to associate with this macro. Each time an instance of this macro is created, this counter is incremented. The incrementing of this counter, of course, resets any "child" counters just like in LaTeX. By default and LaTeX convention, if the macro's first argument is an asterisk (i.e. *), the counter is not incremented.

id
specifies a unique ID for the object. If the object has an associated label (i.e. `\label`), that is its ID. You can also set the ID manually. Otherwise, an ID will be generated based on the result of Python's `id()` function.

idref
a dictionary containing all of the objects referenced by "idref" type arguments. Each idref attribute is stored under the name of the argument in the `idref` dictionary.

level
specifies the hierarchical level of the node in the DOM. For most macros, this will be set to `Node.COMMAND_LEVEL` or `Node.ENVIRONMENT_LEVEL` by the `Command` and `Environment` macros, respectively. However, there are other levels that invoke special processing. In particular, sectioning commands such as `\section` and `\subsection` have levels set to `Node.SECTION_LEVEL` and `Node.SUBSECTION_LEVEL`. These levels assist in the building of an appropriate DOM. Unless you are creating a sectioning command or a command that should act like a paragraph, you should leave the value of this attribute alone. See section 6.3 for more information.

macroName
specifies the name of the LaTeX macro that this class corresponds to. By default, the Python class name is the name that is used, but there are some legal LaTeX macro names that are not legal Python class names. In those cases, you would use `macroName` to specify the correct name. Below is an example.

```
class _illegalname(Command):
    macroName = '@illegalname'
```

Note: This is a class attribute, not an instance attribute.

macroMode
specifies what the current parsing mode is for this macro. Macro classes are instantiated for every invocation including each `\begin` and `\end`. This attribute is set to `Macro.MODE_NONE` for normal commands, `Macro.MODE_BEGIN` for the beginning of an environment, and `Macro.MODE_END` for the end of an environment.

These attributes are used in the `invoke()` method to determine the scope of macros used within the environment. They are also used in printing the source of the macro in the `source` attribute. Unless you really know what you are doing, this should be treated as a read-only attribute.

mathMode
boolean that indicates that the macro is in TeX's "math mode." This is a read-only attribute.

nodeName
the name of the node in the DOM. This will either be the name given in `macroName`, if defined, or the name of the class itself. **Note:** This is a read-only attribute.

ref

specifies the value to return when this macro is referenced (i.e. \ref). This is set automatically when the counter associated with the macro is incremented.

source
specifies the LaTeX source that was parsed to create the object. This is most useful in the renderer if you need to generate an image of a document node. You can simply retrieve the LaTeX source from this attribute, create a LaTeX document including the source, then convert the DVI file to the appropriate image type.

style
specifies style overrides, in CSS format, that should be applied to the output. This object is a dictionary, so style property names are given as the key and property values are given as the values.

```
inst.style['color'] = 'red'
inst.style['background-color'] = 'blue'
```

Note: Not all renderers are going to support CSS styles.

tagName
same as nodeName

title
specifies the title of the current object. If the attributes dictionary contains a title, that object is returned. An AttributeError is thrown if there is no 'title' key in that dictionary. A title can also be set manually by setting this attribute.

digest (*tokens*)
absorb the tokens from the given output stream that belong to the current object. In most commands, this does nothing. However, LaTeX environments have a \begin and an \end that surround content that belong to them. In this case, these environments need to absorb those tokens and construct them into the appropriate document object model (see the Environment class for more information).

digestUntil (*tokens, endclass*)
utility method to help macros like lists and tables digest their contents. In lists and tables, the items, rows, and cells are delimited by \begin and \end tokens. They are simply delimited by the occurrence of another item, row, or cell. This method allows you to absorb tokens until a particular class is reached.

expand()
the expand method is a thin wrapper around the invoke method. The expand method makes sure that all tokens are expanded and will not return a *None* value like invoke.

invoke()
invakes the macro. Invoking the macro, in the general case, includes creating a new context, parsing the options of the macro, and removing the context. LaTeX environments are slightly different. If macroMode is set to Macro.MODE_BEGIN, the new context is kept on the stack. If macroMode is set to Macro.MODE_END, no arguments are parsed, the context is simply popped. For most macros, the default implementation will work fine.

The return value for this method is generally *None* (an empty return statement or simply no return statement). In this case, the current object is simply put into the resultant output stream. However, you can also return a list of tokens. In this case, the returned tokens will be put into the output stream in place of the current object. You can even return an empty list to indicate that you don't want anything to be inserted into the output stream.

locals()
retrieves all of the LaTeX macros that belong to the scope of the current Python based macro.

paragraphs (*force=True*)
group content into paragraphs. Paragraphs are grouped once all other content has been digested. The paragraph grouping routine works like TeX's, in that environments are included inside paragraphs. This is unlike HTML's model, where lists and tables are not included inside paragraphs. The *force* argument allows you to decide whether or not paragraphs should be forced. By default, all content of the node is grouped into paragraphs

whether or not the content originally contained a paragraph node. However, with *force* set to *False*, a node will only be grouped into paragraphs if the original content contained at least one paragraph node.

Even though the paragraph method follow's TEX's model, it is still possible to generate valid HTML content. Any node with the *blockType* attribute set to *True* is considered to be a block-level node. This means that it will be contained in its own paragraph node. This paragraph node will also have the *blockType* attribute set to *True* so that in the renderer the paragraph can be inserted or ignored based on this attribute.

parse(*tex*)

parses the arguments defined in the `args` attribute from the given token stream. This method also calls several hooks as described in the table below.

Method Name	Description
`preParse()`	called at the beginning of the argument parsing process
`preArgument()`	called before parsing each argument
`postArgument()`	called after parsing each argument
`postParse()`	called at the end of the argument parsing process

The methods are called to assist in labeling and counting. For example, by default, the counter associated with a macro is automatically incremented when the macro is parsed. However, if the first argument is a modifier (i.e. *, +, -), the counter will not be incremented. This is handled in the `preArgument()` and `postArgument()` methods.

Each time an argument is parsed, the result is put into the `attributes` dictionary. The key in the dictionary is, of course, the name given to that argument in the `args` string. Modifiers such as *, +, and - are stored under the special key '*modifier*'.

The return value for this method is simply a reference to the `attributes` dictionary.

Note: If `parse()` is called on an instance with `macroMode` set to `Macro.MODE_END`, no parsing takes place.

postArgument(*arg, tex*)

called after parsing each argument. This is generally where label and counter mechanisms are handled.

arg is the Argument instance that holds all argument meta-data including the argument's name, source, and options.

tex is the TeX instance containing the current context

postParse(*tex*)

do any operations required immediately after parsing the arguments. This generally includes setting up the value that will be returned when referencing the object.

preArgument(*arg, tex*)

called before parsing each argument. This is generally where label and counter mechanisms are handled.

arg is the Argument instance that holds all argument meta-data including the argument's name, source, and options.

tex is the TeX instance containing the current context

preParse(*tex*)

do any operations required immediately before parsing the arguments.

refstepcounter(*tex*)

set the object as the current labellable object and increment its counter. When an object is set as the current labellable object, the next `\label` command will point to that object.

stepcounter(*tex*)

step the counter associated with the macro

6.2 plasTeX.ConfigManager — plasTEX Configuration

The configuration system in plasTEX that parses the command-line options and configuration files is very flexible. While many options are setup by the plasTEX framework, it is possible for you to add your own options. This is useful if you have macros that may need to be configured by configurable options, or if you write a renderer that surfaces special options to control it.

The config files that ConfigManager supports are standard INI-style files. This is the same format supported by Python's ConfigParser. However, this API has been extended with some dictionary-like behaviors to make it more Python friendly.

In addition to the config files, ConfigManager can also parse command-line options and merge the options from the command-line into the options set by the given config files. In fact, when adding options to a ConfigManager, you specify both how they appear in the config file as well as how they appear on the command-line. Below is a basic example.

```
from plasTeX.ConfigManager import *
c = ConfigManager()

# Create a new section in the config file.  This corresponds to the
# [ sectionname ] sections in an INI file.  The returned value is
# a reference to the new section
d = c.add_section('debugging')

# Add an option to the 'debugging' section called 'verbose'.
# This corresponds to the config file setting:
#
# [debugging]
# verbose = no
#
d['verbose'] = BooleanOption(
    """ Increase level of debugging information """,
    options = '-v --verbose !-q !--quiet',
    default = False,
)

# Read system-level config file
c.read('/etc/myconfig.ini')

# Read user-level config file
c.read('~/myconfig.ini')

# Parse the current command-line arguments
opts, args = c.getopt(sys.argv[1:])

# Print the value of the 'verbose' option in the 'debugging' section
print c['debugging']['verbose']
```

One interesting thing to note about retrieving values from a ConfigManager is that you get the value of the option rather than the option instance that you put in. For example, in the code above. A BooleanOption in put into the 'verbose' option slot, but when it is retrieved in the print statement at the end, it prints out a boolean value. This is true of all option types. You can access the option instance in the data attribute of the section (e.g. c['debugging'].data['verbose']).

6.2.1 ConfigManager Objects

class ConfigManager(*defaults={ }*)
Instantiate a configuration class for plasTEX that parses the command-line options as well as reads the config files.

The optional argument, *defaults*, is a dictionary of default values for the configuration object. These values are used if a value is not found in the requested section.

__add__(*other*)
merge items from another `ConfigManager`. This allows you to add `ConfigManager` instances with syntax like: config + other. This operation will modify the original instance.

add_section(*name*)
create a new section in the configuration with the given name. This name is the name used for the section heading in the INI file (i.e. the name used within square brackets ([ ]) to start a section). The return value of this method is a reference to the newly created section.

categories()
return the dictionary of categories

copy()
return a deep copy of the configuration

defaults()
return the dictionary of default values

read(*filenames*)
read configuration data contained in files specified by *filenames*. Files that cannot be opened are silently ignored. This is designed so that you can specify a list of potential configuration file locations (e.g. current directory, user's home directory, system directory), and all existing configuration files in the list will be read. A single filename may also be given.

get(*section, option, raw=0, vars={ }*)
retrieve the value of *option* from the section *section*. Setting *raw* to true prevents any string interpolation from occurring in that value. *vars* is a dictionary of addition value to use when interpolating values into the option.

Note: You can alsouse the alternative dictionary syntax: config[section].get(option).

getboolean(*section, option*)
retrieve the specified value and cast it to a boolean

get_category(*key*)
return the title of the given category

getfloat(*section, option*)
retrieve the specified value and cast it to a float

getint(*section, option*)
retrieve the specified value and cast it to and integer

get_opt(*section, option*)
return the option value with any leading and trailing quotes removed

getopt(*args=None, merge=True*)
parse the command-line options. If *args* is not given, the args are parsed from `sys.argv[1:]`. If *merge* is set to false, then the options are not merged into the configuration. The return value is a two element tuple. The first value is a list of parsed options in the form `(option, value)`, and the second value is the list of arguments.

get_optlist(*section, option, delim=','*)
return the option value as a list using *delim* as the delimiter

getraw(*section, option*)

return the raw (i.e. un-interpolated) value of the option

has_category(*key, title*)
add a category to group options when printing the command-line help. Command-line options can be grouped into categories to make options easier to find when printing the usage message for a program. Categories consist of two pieces: 1) the name, and 2) the title. The name is the key in the category dictionary and is the name used when specifying which category an option belongs to. The title is the actual text that you see as a section header when printing the usage message.

has_option(*section, name*)
return a boolean indicating whether or not an option with the given name exists in the given section

has_section(*name*)
return a boolean indicating whether or not a section with the given name exists

__iadd__(*other*)
merge items from another `ConfigManager`. This allows you to add `ConfigManager` instances with syntax like: config += other.

options(*name*)
return a list of configured option names within a section. Options are all of the settings of a configuration file within a section (i.e. the lines that start with 'optionname=').

__radd__(*other*)
merge items from another `ConfigManager`. This allows you to add `ConfigManager` instances with syntax like: other + config. This operation will modify the original instance.

readfp(*fp, filename=None*)
like `read()`, but the argument is a file object. The optional *filename* argument is used for printing error messages.

remove_option(*section, option*)
remove the specified option from the given section

remove_section(*section*)
remove the specified section

__repr__()
return the configuration as an INI formatted string; this also includes options that were set from Python code.

sections()
return a list of all section names in the configuration

set(*section, option, value*)
set the value of an option

__str__()
return the configuration as an INI formatted string; however, do not include options that were set from Python code.

to_string(*source=...*)
return the configuration as an INI formatted string. The *source* option indicates which source of information should be included in the resulting INI file. The possible values are:

Name	Description
COMMANDLINE	set from a command-line option
CONFIGFILE	set from a configuration file
BUILTIN	set from Python code
ENVIRONMENT	set from an environment variable

write(*fp*)
write the configuration as an INI formatted string to the given file object

usage (*categories=[]*)
print the descriptions of all command-line options. If *categories* is specified, only the command-line options from those categories is printed.

6.2.2 ConfigSection Objects

class ConfigSection (*name, data={ }*)
Instantiate a `ConfigSection` object.

name is the name of the section.

data, if specified, is the dictionary of data to initalize the section contents with.

`ConfigSection` objects are rarely instantiated manually. They are generally created using the `ConfigManager` API (either the direct methods or the Python dictionary syntax).

data
dictionary that contains the option instances. This is only accessed if you want to retrieve the real option instances. Normally, you would use standard dictionary key access syntax on the section itself to retrieve the option values.

name
the name given to the section.

copy ()
make a deep copy of the section object.

defaults ()
return the dictionary of default options associated with the parent `ConfigManager`.

get (*option, raw=0, vars={ }*)
retrieve the value of *option*. Setting *raw* to true prevents any string interpolation from occurring in that value. *vars* is a dictionary of addition value to use when interpolating values into the option.

Note: You can alsouse the alternative dictionary syntax: section.get(option).

getboolean (*section, option*)
retrieve the specified value and cast it to a boolean

__getitem__ (*key*)
retrieve the value of an option. This method allows you to use Python's dictionary syntax on a section as shown below.

```
# Print the value of the 'optionname' option
print mysection['optionname']
```

getint (*section, option*)
retrieve the specified value and cast it to and integer

getfloat (*section, option*)
retrieve the specified value and cast it to a float

getraw (*section, option*)
return the raw (i.e. un-interpolated) value of the option

parent
a reference to the parent `ConfigManager` object.

__repr__ ()
return a string containing an INI file representation of the section.

set(*option, value*)
create a new option or set an existing option with the name *option* and the value of *value*. If the given value is already an option instance, it is simply inserted into the section. If it is not an option instance, an appropriate type of option is chosen for the given type.

__setitem__(*key, value*)
create a new option or set an existing option with the name *key* and the value of *value*. This method allows you to use Python's dictionary syntax to set options as shown below.

```
# Create a new option called 'optionname'
mysection['optionname'] = 10
```

__str__()
return a string containing an INI file representation of the section. Options set from Python code are not included in this representation.

to_string([*source*])
return a string containing an INI file representation of the section. The *source* option allows you to only display options from certain sources. See the `ConfigManager.source()` method for more information.

6.2.3 Configuration Option Types

There are several option types that should cover just about any type of command-line and configuration option that you may have. However, in the spirit of object-orientedness, you can, of course, subclass one of these and create your own types. `GenericOption` is the base class for all options. It contains all of the underlying framework for options, but should never be instantiated directly. Only subclasses should be instantiated.

class GenericOption([*docsTring, options, default, optional, values, category, callback, synopsis, environ, registry, mandatory, name, source*])
Declare a command line option.

Instances of subclasses of `GenericOption` must be placed in a `ConfigManager` instance to be used. See the documentation for `ConfigManager` for more details.

docstring is a string in the format of Python documentation strings that describes the option and its usage. The first line is assumed to be a one-line summary for the option. The following paragraphs are assumed to be a complete description of the option. You can give a paragraph with the label 'Valid Values:' that contains a short description of the values that are valid for the current option. If this paragraph exists and an error is encountered while validating the option, this paragraph will be printed instead of the somewhat generic error message for that option type.

options is a string containing all possible variants of the option. All variants should contain the '-', '–', etc. at the beginning. For boolean options, the option can be preceded by a '!' to mean that the option should be turned OFF rather than ON which is the default.

default is a value for the option to take if it isn't specified on the command line

optional is a value for the option if it is given without a value. This is only used for options that normally take a value, but you also want a default that indicates that the option was given without a value.

values defines valid values for the option. This argument can take the following forms:

Type	Description
single value	for `StringOption` this this is a string, for `IntegerOption` this is an integer, for `FloatOption` this is a float. The single value mode is most useful when the value is a regular expression. For example, to specify that a `StringOption` must be a string of characters followed by a digit, 'values' would be set to `re.compile(r'\w+\d')`.
range of values	a two element list can be given to specify the endpoints of a range of valid values. This is probably most useful on `IntegerOption` and `FloatOption`. For example, to specify that an IntegerOption can only take the values from 0 to 10, 'values' would be set to [0,10]. **Note:** This mode must *always* use a Python list since using a tuple means something else entirely.
tuple of values	a tuple of values can be used to specify a complete list of valid values. For example, to specify that an `IntegerOption` can take the values 1, 2, or 3, 'values' would be set to `(1,2,3)`. If a string value can only take the values, 'hi', 'bye', and any string of characters beginning with the letter 'z', 'values' would be set to `('hi','bye',re.compile(r'z.*?'))`. **Note:** This mode must *always* use a Python tuple since using a list means something else entirely.

category is a category key which specifies which category the option belongs to (see the `ConfigManager` documentation on how to create categories).

callback is a function to call after the value of the option has been validated. This function will be called with the validated option value as its only argument.

environ is an environment variable to use as default value instead of specified value. If the environment variable exists, it will be used for the default value instead of the specified value.

registry is a registry key to use as default value instead of specified value. If the registry key exists, it will be used for the default value instead of the specified value. A specified environment variable takes precedence over this value. **Note:** This is not implemented yet.

name is a key used to get the option from its corresponding section. You do not need to specify this. It will be set automatically when you put the option into the `ConfigManager` instance.

mandatory is a flag used to determine if the option itself is required to be present. The idea of a "mandatory option" is a little strange, but I have seen it done.

source is a flag used to determine whether the option was set directly in the `ConfigManager` instance through Python, by a configuration file/command line option, etc. You do not need to specify this, it will be set automatically during parsing. This flag should have the value of *BUILTIN*, *COMMANDLINE*, *CONFIGFILE*, *ENVIRONMENT*, *REGISTRY*, or *CODE*.

`acceptsArgument``()`
return a boolean indicating whether or not the option accepts an argument on the command-line. For example, boolean options do not accept an argument.

`cast``(`*arg*`)`
cast the given value to the appropriate type.

`checkValues``(`*value*`)`
check *value* against all possible valid values for the option. If the value is invalid, raise an `InvalidOptionError` exception.

`clearValue``()`
reset the value of the option as if it had never been set.

`getValue``(`[*default*]`)`
return the current value of the option. If *default* is specified and a value cannot be gotten from any source, it is returned.

__repr__()
return a string containing a command-line representation of the option and its value.

requiresArgument()
return a boolean indicating whether or not the option requires an argument on the command-line.

As mentioned previously, GenericOption is an abstract class (i.e. it should not be instantiated directly). Only subclasses of GenericOption should be instantiated. Below are some examples of use of some of these subclasses, followed by the descriptions of the subclasses themselves.

```
BooleanOption(
   ''' Display help message ''',
   options = '--help -h',
   callback = usage,  # usage() function must exist prior to this
)


BooleanOption(
   ''' Set verbosity ''',
   options = '-v --verbose !-q !--quiet',
)


StringOption(
   '''
   IP address option

   This option accepts an IP address to connect to.

   Valid Values:
   '#.#.#.#' where # is a number from 1 to 255

   ''',
   options = '--ip-address',
   values = re.compile(r'\d{1,3}(\.\d{1,3}){3}'),
   default = '127.0.0.0',
   synopsis = '#.#.#.#',
   category = 'network',  # Assumes 'network' category exists
)


IntegerOption(
   '''
   Number of seconds to wait before timing out

   Valid Values:
   positive integer

   ''',
   options = '--timeout -t',
   default = 300,
   values = [0,1e9],
   category = 'network',
)
```

```
IntegerOption(
   '''
   Number of tries to connect to the host before giving up

   Valid Values:
   accepts 1, 2, or 3 retries

   ''',
   options = '--tries',
   default = 1,
   values = (1,2,3),
   category = 'network',
)


StringOption(
   '''
   Nonsense option for example purposes only

   Valid Values:
   accepts 'hi', 'bye', or any string beginning with the letter 'z'

   ''',
   options = '--nonsense -n',
   default = 'hi',
   values = ('hi', 'bye', re.compile(r'z.*?')),
)
```

class `BooleanOption` ([*`GenericOption` arguments*])

Boolean options are simply options that allow you to specify an 'on' or 'off' state. The accepted values for a boolean option in a config file are 'on', 'off', 'true', 'false', 'yes', 'no', 0, and 1. Boolean options on the command-line do not take an argument; simply specifying the option sets the state to true.

One interesting feature of boolean options is in specifying the command-line options. Since you cannot specify a value on the command-line (the existence of the option indicates the state), there must be a way to set the state to false. This is done using the 'not' operator (!). When specifying the *options* argument of the constructor, if you prefix an command-line option with an exclamation point, the existence of that option indicates a false state rather than a true state. Below is an example of an *options* value that has a way to turn debugging information on (**--debug**) or off (**--no-debug**).

```
BooleanOption( options = '--debug !--no-debug' )
```

class `CompoundOption` ([*`GenericOption` arguments*])

Compound options are options that contain multiple elements on the command-line. They are simply groups of command-line arguments surrounded by a pair of grouping characters (e.g. ( ), [ ], { }, < >). This grouping can contain anything including other command-line arguments. However, all content between the grouping characters is unparsed. This can be useful if you have a program that wraps another program and you want to be able to forward the wrapped program's options on. An example of a compound option used on the command-line is shown below.

```
# Capture the --diff-opts options to send to another program
mycommand --other-opt --diff-opts ( -ib --minimal ) file1 file2
```

class `CountedOption` ([*`GenericOption` arguments*])

A `CountedOption` is a boolean option that keeps track of how many times it has been specified. This is useful for options that control the verbosity of logging messages in a program where the number of times an option is specified, the more logging information is printed.

class `InputDirectoryOption` ([*`GenericOption` arguments*])
An `InputDirectoryOption` is an option that accepts a directory name for input. This directory name is checked to make sure that it exists and that it is readable. If it is not, a `InvalidOptionError` exception is raised.

class `OutputDirectoryOption` ([*`GenericOption` arguments*])
An `OutputDirectoryOption` is an option that accepts a directory name for output. If the directory exists, it is checked to make sure that it is readable. If it does not exist, it is created.

class `InputFileOption` ([*`GenericOption` arguments*])
An `InputFileOption` is an option that accepts a file name for input. The filename is checked to make sure that it exists and is readable. If it isn't, an `InvalidOptionError` exception is raised.

class `OutputFileOption` ([*`GenericOption` arguments*])
An `OutputFileOption` is an option that accepts a file name for output. If the file exists, it is checked to make sure that it is writable. If a name contains a directory, the path is checked to make sure that it is writable. If the directory does not exist, it is created.

class `FloatOption` ([*`GenericOption` arguments*])
A `FloatOption` is an option that accepts a floating point number.

class `IntegerOption` ([*`GenericOption` arguments*])
An `IntegerOption` is an option that accepts an integer value.

class `MultiOption` ([*`GenericOption` arguments,* [*delim, range, template*]])
A `MultiOption` is an option that is intended to be used multiple times on the command-line, or take a list of values. Other options when specified more than once simply overwrite the previous value. `MultiOptions` will append the new values to a list.

The delimiter used to separate multiple values is the comma (,). A different character can be specified in the *delim* argument.

In addition, it is possible to specify the number of values that are legal in the *range* argument. The range argument takes a two element list. The first element is the minimum number of times the argument is required. The second element is the maximum number of times it is required. You can use a '*' (in quotes) to mean an infinite number.

You can cast each element in the list of values to a particular type by using the *template* argument. The *template* argument takes a reference to the option class that you want the values to be converted to.

class `StringOption` ([*`GenericOption` arguments*])
A `StringOption` is an option that accepts an arbitrary string.

6.3 `plasTeX.DOM` — The plasTEX Document Object Model (DOM)

While most LATEX processors use a stream model where the input is directly connected to the output, plasTEX actually works in two phases. The first phase reads in the LATEX document, expands macros, and constructs an object similar to an XML DOM. This object is then passed to the renderer which translates it into the appropriate output format. The benefit to doing it this way is that you are not limited to a single output format. In addition, you can actually apply multiple renderers with only one parse step. This section describes the DOM used by plasTEX, its API, and the similarities and differences between the plasTEX DOM and the XML DOM.

6.3.1 plasTEX vs. XML

The plasTEX DOM and XML DOM have more similarities than differences. This similarity is purely intentional to reduce the learning curve and to prevent reinventing the wheel. However, the XML DOM can be a bit cumbersome especially when you're used to much simpler and more elegant Python code. Because of this, some Python behaviors were adopted into the plasTEX DOM. The good news is that these extensions do not break compatibility with the XML DOM. There are, however, some differences due to conventions used LATEX.

The only significant difference between the plasTEX DOM and the XML DOM is that plasTEX nodes do not have true attributes like in XML. Attributes in XML are more like arguments in LATEX, because they are similar the plasTEX DOM actually puts the LATEX macro arguments into the `attributes` dictionary. This does create an incompatibility though since XML DOM attributes can only be strings whereas LATEX arguments can contain lots of markup. In addition, plasTEX allows you to convert these arguments into Python strings, lists, dictionaries, etc., so essentially any type of object can occur in the `attributes` dictionary.

Other than paying attention to the the attributes dictionary difference, you can use most other XML DOM methods on plasTEX document objects to create nodes, delete nodes, etc. The full API is described below.

In most cases, you will not need to be concerned with instantiating nodes. The plasTEX framework does this. However, the API can be helpful if you want to modify the document object that plasTEX creates.

6.3.2 Node Objects

class Node()
The `Node` class is the base class for all nodes in the plasTEX DOM inluding elements, text, etc.

attributes
a dictionary containing the attributes, in the case of plasTEX the LATEX macro arguments

childNodes
a list of the nodes that are contained by this one. In plasTEX, this generally contains the contents of a LATEX environment.

isElementContentWhitespace
boolean indicating whether or not the node only contains whitespace.

lastChild
the last node in the `childNodes` list. If there are no child nodes, the value is *None*.

nodeName
the name of the node. This is either the special node name as specified in the XML DOM (e.g. #document-fragment, #text, etc.), or, if the node corresponds to an element, it is the name of the element.

nodeType
integer indicating the type of the node. The node types are defined as:

- `Node.ELEMENT_NODE`
- `Node.ATTRIBUTE_NODE`
- `Node.TEXT_NODE`
- `Node.CDATA_SECTION_NODE`
- `Node.ENTITY_REFERENCE_NODE`
- `Node.ENTITY_NODE`
- `Node.PROCESSING_INSTRUCTION_NODE`
- `Node.COMMENT_NODE`
- `Node.DOCUMENT_NODE`
- `Node.DOCUMENT_TYPE_NODE`

- `Node.DOCUMENT_FRAGMENT_NODE`
- `Node.NOTATION_NODE`

Note: These are defined by the XML DOM, not all of them are used by plasTEX.

`parentNode`
refers to the node that contains this node

`previousSibling`
the node in the document that is adjacent to and immediately before this node. If one does not exist, the value is *None*.

`nextSibling`
the node in the document that is adjacent to and immediately after this node. If one does not exist, the value is *None*.

`ownerDocument`
the node that owner of, and ultimate parent of, all nodes in the document

`textContent`
contains just the text content of this node

`unicode`
specifies a unicode string that could be used in place of the node. This unicode string will be converted into tokens in the plasTEX output stream.

`userdata`
dictionary used for holding user-defined data

`__add__`(*other*)
create a new node that is the sum of *self* and *other*. This allows you to use nodes in Python statements like: node + other.

`append`(*newChild*)
adds a new child to the end of the child nodes

`appendChild`(*newChild*)
same as `append`

`cloneNode`(*deep=False*)
create a clone of the current node. If *deep* is true, then the attributes and child nodes are cloned as well. Otherwise, all references to attributes and child nodes will be shared between the nodes.

`__cmp__`(*other*)
same as `isEqualNode`, but allows you to compare nodes using the Python statement: node == other.

`extend`(*other*)
appends *other* to list of children then returns *self*

`__getitem__`(*i*)
returns the child node at the index given by *i*. This allows you to use Python's slicing syntax to retrieve child nodes: node[i].

`getUserData`(*key*)
retrieves the data in the `userdata` dictionary under the name *key*

`hasAttributes`()
returns a boolean indicating whether or not this node has attributes defined

`hasChildNodes`()
returns a boolean indicating whether or not the node has child nodes

`__iadd__`(*other*)
same as `extend`. This allows you to use nodes in Python statements like: node += other.

insert(*i, newChild*)
inserts node *newChild* into position *i* in the child nodes list

insertBefore(*newChild, refChild*)
inserts *newChild* before *refChild* in this node. If *refChild* is not found, a `NotFoundErr` exception is raised.

isEqualNode(*other*)
indicates whether the given node is equivalent to this one

isSameNode(*other*)
indicates whether the given node is the same node as this one

__iter__()
returns an iterator that iterates over the child nodes. This allows you to use Python's `iter()` function on nodes.

__len__()
returns the number of child nodes. This allows you to use Python's `len()` function on nodes.

normalize()
combine consecutive text nodes and remove comments in this node

pop(*index=-1*)
removes child node and the index given by *index*. If no index is specified, the last child is removed.

__radd__(*other*)
create a new node that is the sum of *other* and *self*. This allows you to use nodes in Python statements like: other + node.

replaceChild(*newChild, oldChild*)
replaces *oldChild* with *newChild* in this node. If *oldChild* is not found, a `NotFoundErr` exception is raised.

removeChild(*oldChild*)
removes *oldChild* from this node. If *oldChild* is not found, a `NotFoundErr` exception is raised.

__setitem__(*i, node*)
sets the item at index *i* to *node*. This allows you to use Python's slicing syntax to insert child nodes; see the example below.

```
mynode[5] = othernode
mynode[6:10] = [node1, node2]
```

setUserData(*key, data*)
put data specified in *data* into the `userdata` dictionary under the name given by *key*

toXML()
return an XML representation of the node

6.3.3 DocumentFragment Objects

class DocumentFragment()
A collection of nodes that make up only part of a document. This is mainly used to hold the content of a LaTeX macro argument.

6.3.4 Element Objects

class Element()
The base class for all element-type nodes in a document. Elements generally refer to nodes created by LaTeX commands and environments.

getAttribute(*name*)
returns the attribute specified by *name*

getElementById(*elementId*)
retrieve the element with the given ID

getElementsByTagName(*tagName*)
retrieve all nodes with the given name in the node

hasAttribute(*name*)
returns a boolean indicating whether or not the specified attribute exists

removeAttribute(*name*)
removes the attribute *name* from the `attributes` dictionary

setAttribute(*name, value*)
sets the attribute *value* in the `attributes` dictionary using the key *name*

6.3.5 Text Objects

class Text()
This is the node type used for all text data in a document object. Unlike XML DOM text nodes, text nodes in plasTEX are not mutable. This is because they are a subclass of *unicode*. This means that they will respond to all of the standard Python string methods in addition to the `Node` methods and the methods described below.

data
the text content of the node

length
the length of the text content

nodeValue
the text content of the node

wholeText
returns the text content from the current text node as well as its siblings

6.3.6 Document Objects

class Document()
The top-level node of a document that contains all other nodes.

createDocumentFragment()
instantiate a new document fragment

createElement(*tagName*)
instantiate a new element with the given name

createTextNode(*data*)
instantiate a new text node initialized with *data*

importNode(*importedNode, deep=False*)
import a node from another document. If *deep* is true, all nodes within *importedNode* are cloned.

normalizeDocument()
concatenate all consecutive text nodes and remove comments

6.3.7 Command Objects

class Command()
The Command class is a subclass of Macro. This is the class that should be subclassed when creating Python based macros that correspond to LaTeX commands.

For more information on the Command class' API, see the Macro class.

6.3.8 Environment Objects

class Environment()
The Environment class is a subclass of Macro. This is the class that should be subclassed when creating Python based macros that correspond to LaTeX environments. The main difference between the processing of Commands and Environments is that the invoke() method does special handling of the LaTeX document context, and the digest() method absorbs the output stream tokens that are encapsulated by the \begin and \end tokens.

For more information on the Environment class' API, see the Macro class.

6.3.9 TeXFragment Objects

class TeXFragment()
A fragment of a document. This class is used mainly to store the contents of LaTeX macro arguments.

source
the LaTeX source representation of the document fragment

6.3.10 TeXDocument Objects

class TeXDocument()
A complete LaTeX document.

charsubs
a list of two element tuples containing character substitutions for all text nodes in a document. This is used to convert charcter strings like "---" into "—". The first element in each tuple in the string to replace, the second element is the unicode character or sequence to replace the original string with.

preamble
returns the LaTeX source representation of the document preamble (i.e. everything before the \begin{document})

source
the LaTeX source representation of the document

6.4 plasTeX.TeX — The TeX Stream

The TeX stream is the piece of plasTeX where the parsing of the LaTeX document takes place. While the TeX class is fairly large, there are only a few methods and attributes designated in the public API.

The TeX stream is based on a Python generator. When you feed it a LaTeX source file, it processes the file much like TeX itself. However, on the output end, rather than a DVI file, you get a plasTeX document object. The basic usage is shown in the code below.

```
from plasTeX.TeX import TeX
doc = TeX(file='myfile.tex').parse()
```

6.4.1 TeX Objects

class TeX([*ownerDocument, file*])

The TeX class is the central TEX engine that does all of the parsing, invoking of macros, and other document building tasks. You can pass in an owner document if you have a customized document node, or if it contains a customized configuration; otherwise, the default TeXDocument class is instantiated. The *file* argument is the name of a LATEX file. This file will be searched for using the standard LATEX technique and will be read using the default input encoding in the document's configuration.

disableLogging()

disables logging. This is useful if you are using the TeX object within another library and do not want all of the status information to be printed to the screen.

Note: This is a class method.

filename

the current filename being processed

jobname

the name of the basename at the top of the input stack

lineNumber

the line number of the current file being processed

expandTokens(*tokens, normalize=False*)

expand a list of unexpanded tokens. This method can be used to expand tokens without having them sent to the output stream. The returned value is a TeXFragment populated with the expanded tokens.

input(*source*)

add a new input source to the input stack. *source* should be a Python file object. This can be used to add additional input sources to the stream after the TeX object has been instantiated.

__iter__()

return a generator that iterates through the tokens in the source. This method allows you to treat the TeX stream as an iterable and use it in looping constructs. While the looping is generally handled in the parse() method, you can manually expand the tokens in the source by looping over the TeX object as well.

```
for tok in TeX(open('myfile.tex')):
    print tok
```

itertokens()

return an iterator that iterates over the unexpanded tokens in the input document.

kpsewhich(*name*)

locate the given file in a kpsewhich-like manner. The full path to the file is returned if it is found; otherwise, *None* is returned. **Note:** Currently, only the directories listed in the environment variable TEXINPUTS are searched.

normalize(*tokens*)

joins consecutive text tokens into a string. If the list of tokens contain tokens that are not text tokens, the original list of tokens is returned.

parse(*output=None*)

parse the sources currently in the input stack until they are empty. The *output* argument is an optional

`Document` node to put the resulting nodes into. If none is supplied, a `TeXDocument` instance will be created. The return value is the document from the *output* argument or the instantiated `TeXDocument` object.

`pushToken`(*token*)
pushes a token back into the input stream to be re-read.

`pushTokens`(*tokens*)
pushes a list of tokens back into the input stream to be re-read.

`readArgument`(**args, **kwargs*)
parse a macro argument without the LaTeX source that created it. This method is just a thin wrapper around `readArgumentAndSource`. See that method for more information.

`readArgumentAndSource`(*spec=None, subtype=None, delim=',', expanded=False, default=None, parentNode=None, name=None*)
parse a macro argument. Return the argument and the LaTeX source that created it. The arguments are described below.

Option	Description
spec	string containing information about the type of argument to get. If it is 'None', the next token is returned. If it is a two-character string, a grouping delimited by those two characters is returned (i.e. '[]'). If it is a single-character string, the stream is checked to see if the next character is the one specified. In all cases, if the specified argument is not found, 'None' is returned.
type	data type to cast the argument to. New types can be added to the self.argtypes dictionary. The key should match this 'type' argument and the value should be a callable object that takes a list of tokens as the first argument and a list of unspecified keyword arguments (i.e. **kwargs) for type specific information such as list delimiters.
subtype	data type to use for elements of a list or dictionary
delim	item delimiter for list and dictionary types
expanded	boolean indicating whether the argument content should be expanded or just returned as an unexpanded text string
default	value to return if the argument doesn't exist
parentNode	the node that the argument belongs to
name	the name of the argument being parsed

The return value is always a two-element tuple. The second value is always a string. However, the first value can take the following values.

Value	Condition
None	the requested argument wasn't found
object of requested type	if *type* was specified
list of tokens	all other arguments

`source`(*tokens*)
return the LaTeX representation of the tokens in *tokens*

`textTokens`(*text*)
convert a string of text into a series of tokens

6.5 `plasTeX.Context` — The TeX Context

The `Context` class stores all of the information associated with the currently running document. This includes things like macros, counters, labels, references, etc. The context also makes sure that localized macros get popped off when processing leaves a macro or environment. The context of a document also has the power to create new counters, dimens, if commands, macros, as well as change token category codes.

Each time a `TeX` object is instantiated, it will create its own context. This context will load all of the base macros and initialize all of the context information described above.

6.5.1 Context Objects

class `Context` ([*load*])
Instantiate a new context.

If the *load* argument is set to true, the context will load all of the base macros defined in plasTEX. This includes all of the macros used in the standard TEX and LATEX distributions.

`contexts`
stack of all macro and category code collections currently in the document being processed. The item at index 0 include the global macro set and default category codes.

`counters`
a dictionary of counters.

`currentlabel`
the object that is given the label when a `\label` macro is invoked.

`isMathMode`
boolean that specifies if we are currently in TEX's math mode or not.

`labels`
a dictionary of labels and the objects that they refer to.

`addGlobal` (*key, value*)
add a macro *value* with name *key* to the global namespace.

`addLocal` (*key, value*)
add a macro *value* with name *key* to the current namespace.

`append` ([*context*])
same as `push()`

`catcode` (*char, code*)
set the category code for a character in the current scope. *char* is the character that will have its category code changed. *code* is the TEX category code (0-15) to change it to.

`chardef` (*name, num*)
create a new TEX chardef like `\chardef`.

name is the name of the command to create.

num is the character number to use.

`__getitem__` (*key*)
look through the stack of macros and return the one with the name *key*. The return value is an *instance* of the requested macro, not a reference to the macro class. This method allows you to use Python's dictionary syntax to retrieve the item from the context as shown below.

```
tex.context['section']
```

`importMacros` (*context*)
import macros from another context into the global namespace. The argument, *context*, must be a dictionary of macros.

`label` (*label*)
set the given label to the currently labelable object. An object can only have one label associated with it.

let (*dest, source*)
create a new TEX let like \let.

dest is the command sequence to create.

source is the token to set the command sequence equivalent to.

Example

```
c.let('bgroup', BeginGroup('{'))
```

loadBaseMacros ()
imports all of the base macros defined by plasTEX. This includes all of the macros specified by the TEX and LATEX systems.

loadLanguage (*language, document*)
loads a language package to configure names such as \figurename, \tablename, etc. See Section 6.5.2 for more information.

language is a string containing the name of the language file to load.

document is the document object being processed.

loadINIPackage (*inifile*)
load an INI formatted package file (see section 4.3 for more information).

loadPackage (*tex, file,* [*options*])
loads a LATEX package.

tex is the TEX processor to use in parsing the package content

file is the name of the package to load

options is a dictionary containing the options to pass to the package. This generally comes from the optional argument on a \usepackage or \documentclass macro.

The package being loaded by this method can be one of three type: 1) a native LATEX package, 2) a Python package, or 3) an INI formatted file. The Python version of the package is searched for first. If it is found, it is loaded and an INI version of the package is also loaded if it exists. If there is no Python version, the true LATEX version of the package is loaded. If there is an INI version of the package in the same directory as the LATEX version, that file is loaded also.

newcommand (*name*[, *nargs*[, *definition*[, *opt*]]])
create a new LATEX command like \newcommand.

name is the name of the macro to create.

nargs is the number of arguments including optional arguments.

definition is a string containing the macro definition.

opt is a string containing the default optional value.

Examples

```
c.newcommand('bold', 1, r'\\textbf{#1}')
c.newcommand('foo', 2, r'{\\bf #1#2}', opt='myprefix')
```

newcount (*name*[, *initial*])
create a new count like \newcount.

newcounter (*name,* [*resetby, initial, format*])
create a new counter like \newcounter.

name is the name of the counter to create.

resetby is the counter that, when incremented, will reset the new counter.

initial is the initial value for the counter.

format is the printed format of the counter.

In addition to creating a new counter macro, another macro corresponding to the \the*name* is created which prints the value of the counter just like in LaTeX.

newdef (*name*[, *args*[, *definition*[, *local*]]])
create a new TeX definition like \def.

name is the name of the definition to create.

args is a string containing the TeX argument profile.

definition is a string containing the macro code to expand when the definition is invoked.

local is a boolean that specifies that the definition should only exist in the local scope. The default value is true.

Examples

```
c.newdef('bold', '#1', '{\\bf #1}')
c.newdef('put', '(#1,#2)#3', '\\dostuff{#1}{#2}{#3}')
```

newdimen (*name*[, *initial*])
create a new dimen like \newdimen.

newenvironment (*name*[, *nargs*[, *definition*[, *opt*]]])
create a new LaTeX environment like \newenvironment. This works exactly like the newcommand() method, except that the *definition* argument is a two element tuple where the first element is a string containing the macro content to expand at the \begin, and the second element is the macro content to expand at the \end.

Example

```
c.newenvironment('mylist', 0, (r'\\begin{itemize}', r'\\end{itemize}'))
```

newif (*name*[, *initial*])
create a new if like \newif. This also creates macros corresponding to *name*true and *name*false.

newmuskip (*name*[, *initial*])
create a new muskip like \newmuskip.

newskip (*name*[, *initial*])
create a new skip like \newskip.

packages
a dictionary of LaTeX packages. The keys are the names of the packages. The values are dictionaries containing the options that were specified when the package was loaded.

pop ([*obj*])
pop the top scope off of the stack. If *obj* is specified, continue to pop scopes off of the context stack until the scope that was originally added by *obj* is found.

push ([*context*])
add a new scope to the stack. If a macro instance *context* is specified, the new scope's namespace is given by that object.

ref (*obj, label*)
set up a reference for resolution.

obj is the macro object that is doing the referencing.

label is the label of the node that *obj* is looking for.

If the item that *obj* is looking for has already been labeled, the idref attribute of *obj* is set to the abject. Otherwise, the reference is stored away to be resolved later.

setVerbatimCatcodes()
set the current set of category codes to the set used for the verbatim environment.

whichCode(*char*)
return the character code that *char* belongs to. The category codes are the same codes used by TEX and are defined in the `Token` class.

6.5.2 Context language

Contexts objects hold language information for the currently running document. The current language is stored in *Context.currentLanguage*. It can be changed from the TEX source using the babel package which invokes the *Context.loadLanguage* method. New terms can be added in a user defined language file using the lang-terms options (see Section 2.1.2). Languages files are xml files. The following example should be self-explanatory.

```
<languages>
  <terms lang="fr" babel="french">
    <term name="proof">DÃl'monstration</term>
  </terms>
</languages>
```

This allows to add new terms which are then available to renderers in the dictionary *Context.terms*. It also allows to override default translations. For instance the above language file overwrites the default translation of "proof" as "Preuve" in French.

6.6 `plasTeX.Renderers` — The plasTEX Rendering Framework

The renderer is responsible for taking the information in a plasTEX document object and creating a another (usually visual) representation of it. This representation may be HTML, XML, RTF, etc. While this could be implemented in various ways. One rendering framework is included with plasTEX.

The renderer is essentially just a dictionary of functions[1]. The keys in this dictionary correspond to names of the nodes in the document object. The values are the functions that are called when a node in the document object needs to be rendered. The only argument to the function is the node itself. What this function does in the rendering process is completely up to it; however, it should refrain from changing the document object itself as other renderers may be using that same object.

There are some responsibilities that all renderers share. Renderers are responsible for checking options in the configuration object. For instance, renderers are responsible for generating filenames, creating directories, writing files in the proper encoding, generating images, splitting the document into multiple output files, etc. Of course, how it accomplishes this is really renderer dependent. An example of a renderer based on Zope Page Templates (ZPT) is included with plasTEX. This renderer is capable of generating XML and HTML output.

6.6.1 Renderer Objects

class Renderer()
Base class for all renderers. `Renderer` is a dictionary and contains functions that are called for each node in the plasTEX document object. The keys in the dictionary correspond to the names of the nodes.

This renderer implementation uses a mixin called `Renderable` that is mixed into the `Node` class prior to rendering. `Renderable` adds various methods to the `Node` namespace to assist in the rendering process. The primary inclusion

[1] "functions" is being used loosely here. Actually, any Python callable object (i.e. function, method, or any object with the `__call__` method implemented) can be used

is the `__unicode__()` method. This method returns a unicode representation of the current node and all of its child nodes. For more information, see the `Renderable` class documentation.

default
: the default renderer value. If a node is being rendered and no key in the renderer matches the name of the node being rendered, this function is used instead.

fileExtension
: contains the file extension to use for generated files. This extension is only used if the filename generator does not supply a file extension.

files
: a list of files created during rendering.

imageAttrs
: contains a string template that renders the placeholder for the image attributes: width, height, and depth. This placeholder is inserted into the document where the width, height, and depth of an image is needed. The placeholder is needed because images are not generated until after the document is rendered. See the `Imager` API (section 6.7) for more information.

imageUnits
: contains a string template that renders the placeholder for the image attribute units. This placeholder is inserted in the document any time an attribute of a particular unit is requested. This placeholder will always occur immediately after the string generated by `imageAttrs`. The placeholder is needed because images are not generated until after the document is rendered. See the `Imager` API (section 6.7) for more information.

imager
: a reference to an `Imager` implementation. Imagers are responsible for generating images from LaTeX code. This is needed for output types which aren't capable of displaying equations, LaTeX pictures, etc. such as HTML.

imageTypes
: contains a list of file extensions of valid image types for the renderer. The first element in the list is the default image format. This format is used when generating images (if the image type isn't specified by the filename generater). When static images are simply copied from the document, their format is checked against the list of supported image types. If the static image is not in the correct format it is converted to the default image format. Below is an example of a list of image types used in the HTML renderer. These image types are valid because web browsers all support these formats.

```
imageTypes = ['.png','.gif','.jpg','.jpeg']
```

vectorImageTypes
: contains a list of file extensions of valid vector image types for the renderer. The first element in the list is the default vector image format. This format is used when generating images. Static images are simply copied into the output document directory. Below is an example of a list of image types used in the HTML renderer. These image types are valid because there are plug-ins available for these formats.

```
vectorImageTypes = ['.svg']
```

newFilename
: filename generator. This method generates a basename based on the options in the configuration.

  The generator has an attribute called `namespace` which contains the namespace used to resolve the variables in the filename string. This namespace should be populated prior to invoking the generator. After a successful filename is generated, the namespace is automatically cleared (with the exception of the variables sent in the namespace when the generator was instantiated).

  Note: This generator can be accessed in the usual generator fashion, or called like a function.

outputType

a function that converts the content returned from each rendered node to the appropriate value.

textDefault
the default renderer to use for text nodes.

cleanup(*document, files*[, *postProcess*])
this method is called once the entire rendering process is finished. Subclasses can use this method to run any post-rendering cleanup tasks. The first argument, *document*, is the document instance that is being rendered. The second argument, *files*, is a list of all of the filenames that were created.

This method opens each file, reads the content, and calls `processFileContent` on the file content. It is suggested that renderers override that method instead of `cleanup`.

In addition to overriding `processFileContent`, you can post-process file content without having to subclass a renderer by using the *postProcess* argument. See the `render` method for more information.

find(*keys*[, *default*])
locate a rendering method from a list of possibilities.

keys is a list of strings containing the requested name of a rendering method. This list is traversed in order. The first renderer that is found is returned.

default is a default rendering method to return if none of the keys exists in the renderer.

initialize()
this routine is called after the renderer is instantiated. It can be used by subclasses to do any initialization routines before the rendering process.

processFileContent(*document, content*)
post-processing routine that allows renders to modify the output documents one last time before the rendering process is finished. *document* is the input document instance. *content* is the content of the file in a unicode object. The value returned from this method will be written to the output file in the appropriate encoding.

render(*document*[, *postProcess*])
invokes the rendering process on *document*. You can post-process each file after it is rendered by passing a function into the *postProcess* argument. This function must take two arguments: 1) the document object and 2) the content of a file as a unicode object. It should do whatever processing it needs to the file content and return a unicode object.

6.6.2 Renderable MixIn

class Renderable()
The `Renderable` mixin is mixed into the `Node` namespace prior to the rendering process. The methods mixed in assist in the rendering process.

filename
the filename that this object will create. Objects that don't create new files should simply return *None*. The configuration determines which nodes should create new files.

image
generate an image of the object and return the image filename. See the `Imager` documentation in section 6.7 for more information.

vectorImage
generate a vector image of the object and return the image filename. See the `Imager` documentation in section 6.7 for more information.

url
return the relative URL of the object.

If the object actually creates a file, just the filename will be returned (e.g. 'foo.html'). If the object is within a file, both the filename and the anchor will be returned (e.g. 'foo.html#bar').

__str__()
same as __unicode__().

__unicode__()
invoke the rendering process on all of the child nodes. The rendering process includes walking through the child nodes, looking up the appropriate rendering method from the renderer, and calling the method with the child node as its argument.

In addition to the actual rendering process, this method also prints out some status information about the rendering process. For example, if the node being rendered has a non-empty `filename` attribute, that means that the node is generating a new file. This filename information is printed to the log. One problem with this methodology is that the filename is not actually created at this time. It is assumed that the rendering method will check for the `filename` attribute and actually create the file.

6.7 `plasTeX.Imagers` — The plasTEX Imaging Framework

The imager framework is used when an output format is incapable of representing part of a LATEX document natively. One example of this is equations in HTML. In cases like this you can use an `Imager` to generate images of the commands and environments that cannot be rendered in any other way.

Currently, plasTEX comes with several imager implementations based on **dvi2bitmap** (`http://dvi2bitmap.sourceforge.net/`), **dvipng** (`http://savannah.nongnu.org/projects/dvipng/`), and **ghostscript** with the PNG driver (`http://www.cs.wisc.edu/~ghost/doc/GPL/index.htm`) called gspdfpng and gspspng, as well as one that uses OS X's CoreGraphics library. Creating imagers based on other programs is quite simple, and more are planned for future releases.

In addition to imagers that generate bitmap images, it is also possible to generate vector images using programs like dvisvg (`http://dvisvg.sourceforge.net/`) or dvisvgm (`http://dvisvgm.sourceforge.net/`).

The `Imager` framework does all of its work in temporary directories the one requirement that it has is that `Imager` subclasses need to generate images with the basenames 'img%d' where '%d' is the number of the image.

The only requirement by the plasTEX framework is that the imager class within the imager module is called "Imager" and should be installed in the `plasTeX.Imagers` package. The basename of the imager module is the name used when plasTEX looks for a specified imager.

6.7.1 Imager Objects

class Imager(*document*)
Instantiate the imager class.

document the document object that is being rendered.

The `Imager` class is responsible for creating a LATEX document of requested images, compiling it, and generating images from each page in the document.

command
specifies the converter that translates the output from the LATEX document compiler (e.g. PDF, DVI, PS) into images (e.g. PNG, JPEG, GIF). The only requirement is that the basename of each image is of the form 'img%d' where '%d' is the number of the image.

Note: This is a class attribute.

Writing a renderer requires you to at least override the command that creates images. It can be as simple as the example below.

```
import plasTeX.Imagers
class DVIPNG(plasTeX.Imagers.Imager):
    """ Imager that uses dvipng """
    command = 'dvipng -o img%d.png -D 110'
```

compiler
specifies the LaTeX document compiler (i.e. latex, pdflatex) command.

Note: This is a class attribute.

config
contains the "images" section of the document configuration.

fileExtension
contains the file extension to use if no extension is supplied by the filename generator.

imageAttrs
contains a string template that will be used as a placeholder in the output document for the image height, width, and depth. These attributes cannot be determined in real-time because images are not generated until after the document has been fully rendered. This template generates a string that is put into the output document so that the image attributes can be post-processed in. For example, the default template (which is rather XML/HTML biased) is:

```
&${filename}-${attr};
```

The two variables available are *filename*, the filename of the image, and *attr*, the name of the attr (i.e. width, height, or depth).

imageUnits
contains a string template that will be used as a placeholder in the output document for the image units. This template generates a string that is put into the output document so that the image attribute units can be post-processed in. For example, the default template (which is rather XML/HTML biased) is:

```
&${units);
```

The only variable available is *units* and contains the CSS unit that was requested. The generate string will always occur immediately after the string generated by `imageAttrs`.

images
dictionary that contains the `Image` objects corresponding to the requested images. The keys are the image filenames.

newFilename
callable iterator that generates filenames according to the filename template in the configuration.

source
file object where the image LaTeX document is written to.

verification
command that verifies the existence of the image converter on the current machine. If `verification` is not specified, the executable specified in `command` is executed with the **--help**. If the return code is zero, the imager is considered valid. If the return code is anything else, the imager is not considered valid.

close()
closes the generated LaTeX document and starts the image generation routine.

compileLatex(*source*)
the method responsible for compiling the LaTeX source.

source is a file object containing the LaTeX document.

convert (*output*)
sets up the temporary environment for the image converter, then executes `executeConverter`. It also moves the generated images into their final location specified in the configuration.

executeConverter (*output*)
executes the command that converts the output from the LaTeX compiler into image files.

output is a file object containing the compiled output of the LaTeX document.

getImage (*node*)
get an image for *node* in any way possible. The node is first checked to see if the `imageoverride` attribute is set. If it is, that image is copied to the image directory. If `imageoverride` is not set, or there was a problem in saving the image in the correct format, an image is generated using the source of *node*.

newImage (*text,* [*context, filename*])
invokes the creation of an image using the LaTeX content in *text*.

context is the LaTeX code that sets up the context of the document. This generally includes the setting of counters so that counters used within the image code are correct.

filename is an optional filename for the output image. Generally, image filenames are generated automatically, but they can be overridden with this argument.

verify ()
verifies that the command in `command` is valid for the current machine. The `verify` method returns *True* if the command will work, or *False* if it will not.

writeImage (*filename, code, context*)
writes the LaTeX code to the generated document that creates the image content.

filename is the final filename of the image. This is not actually used in the document, but can be handy for debugging.

code is the LaTeX code that an image is needed of.

context is the LaTeX code that sets up the context of the document. This generally includes the setting of counters so that counters used within the image code are correct.

writePreamble (*document*)
this method is called when the imager is instantiated and is used to write any extra information to the preamble. If overridden, the subclass needs to make sure that *document.preamble.source* is the first thing written to the preamble.

6.7.2 Image Objects

class Image (*filename, config,* [*width, height, alt, depth, longdesc*])
Instantiate an `Image` object.

Image objects contain information about the generated images. This information includes things such as width, height, filename, absolute path, etc. Images objects also have the ability to crop the image that they reference and return information about the baseline of the image that can be used to properly align the image with surrounding text.

filename is the input filename of the image.

config is the "images" section of the document configuration.

width is the width of the image. This is usually extracted from the image file automatically.

height is the height of the image. This is usually extracted from the image file automatically.

alt is a text alternative of the image to be use by renderers such as HTML.

depth is the depth of the image below the baseline of the surrounding text. This is generally calculated automatically when the image is cropped.

longdesc is a long description used to describe the content of the image for renderers such as HTML.

alt
: a text alternative of the image to be use by renderers such as HTML.

config
: the “images” section of the document’s configuration.

depth
: the depth of the image below the baseline of the surrounding text. This is generally calculated automatically when the image is cropped.

filename
: the filename of the image.

height
: the heigt of the image in pixels.

longdesc
: a long description used to describe the content of the image for renderers such as HTML.

path
: the absolute path of the image file.

url
: the URL of the image. This may be used during rendering.

width
: the width of the image in pixels.

crop()
: crops the image so that the image edges are flush with the image content. It also sets the `depth` attribute of the image to the number of pixels that the image extends below the baseline of the surrounding text.

APPENDIX

A

About This Document

This document was writted using LaTeX (`http://www.latex-project.org/`). The documents use macros written for documenting the Python (`http://www.python.org`) language and Python packages. Generating the PDF version of the document is simply a matter of using the **pdflatex** command. Generating the HTML version of the document, of course, uses plasTeX.

The wonderful thing about the HTML version is that it was generated from the LaTeX source and Python style files without customization[1]! In fact, in its current state, plasTeX can generate the HTML versions of the Python documentation found on their website, `http://www.python.org/doc/`. Without customization of plasTeX, the only remaining issues are that the module index is missing and there are some formatting differences. Not bad, considering plasTeX is doing actually expanding the LaTeX document natively.

[1] Ok, there was one customization to `\var` for a whitespace issue, but the change works both in the PDF and HTML version

APPENDIX

B

Frequently Asked Questions

B.1 Parsing LaTeX

B.1.1 How can I make plasTeX work with my complicated macros?

While plasTeX makes a valiant effort to expand all LaTeX macros, it isn't TeX and may have problems if your macros are complicated. There are things that you can do to remedy the situation. If you are getting failures or warnings, you can do one of two things: 1) you can create a simplified version of the macro that plasTeX uses for its work, while LaTeX uses the more complicated one, or 2) you can implement the macro as a Python class.

In the first solution, you can use the `\ifplastex` construct to wrap your plasTeX and LaTeX versions of the macros. You can even just remove parts of the macros. See the example below.

```
% Print a double line, then bold the text.
% In plasTeX, leave the lines out.
\newcommand{\mymacro}[1]{\ifplastex\else\vspace*{1in}\fi\textbf{#1}}
```

Depending on how complicated you macro is, you may want to implement it as a Python class instead of a LaTeX macro. Using a Python class gives you full access to all of the plasTeX internal mechanisms to do whatever you need to do in your macro. To read more about writing Python class macros, see the section 4.

B.1.2 How can I get plasTeX to find my LaTeX packages?

There are two types of packages that can be loaded by plasTeX: 1) native LaTeX packages, and 2) packages written entirely in Python. plasTeX first looks for packages written in Python. Packages such as this are written specifically for plasTeX and will yield better parsing performance as well as better looking output. Python-based packages are valid Python packages as well. So to load them, you must add the directory where your Python packages are to your PYTHONPATH environment variable. For more information about Python-based packages, see the section 4.3.

If you have a true LaTeX package, plasTeX will try to locate it using the **kpsewhich** program just like LaTeX does.